★piler

Not a VM to rule no one

Francesco Bertolotti

Id. Number: R12898

Scuola di Dottorato in Informatica
PhD in Computer Science

PhD School Headmaster: Prof. Roberto Sassi

Advisor: Prof. Walter Cazzola

UNIVERSITÀ DEGLI STUDI DI MILANO
Computer Science Department
ADAPT-Lab

Ciclo XXXVI
INF/01 Informatica
Academic Year 2022-2023

COMPUTER SCIENCE DEPARTMENT • ADAPT-LAB

Accept & go to conflict
—an error log.

Contents

Preface

These past three years have been an incredible journey, filled with both memorable and challenging moments. Throughout this period, I had the privilege of immersing myself in topics that truly captivated my interest. On the other hand, now that I need to find Ariadne's thread of my work, I am a little lost. I extensively thought about this matter for the last year of my Ph.D. I have come to an unconventional conclusion, I do not wish my thesis to be the summary of my works. Instead, I want to dive deep into the subject that occupied most of my last year as a Ph.D. student—the ★piler (Starpiler, a name that, let's admit, sounds undeniably cool). Nonetheless, in order to satisfy my advisor and the Ph.D. commission, I will make every effort to establish connections between the various projects I have developed or contributed to throughout these years.

To the reader who has chosen to delve into this thesis, I extend my gratitude. It is a rare occasion for someone to dedicate their time to immerse themselves in the technical intricacies of compilers. Should you ever feel disoriented during this modest work or simply wish to reach out to the author (surely, to share your appreciation for the excellence of this thesis), please feel free to send an email to francesco.bertolotti@unimi.it. Your feedback and correspondence are sincerely welcomed.

I would also like to take a moment to express my heartfelt gratitude to the individuals who have been crucial to my mental stability throughout these years. Firstly, I extend my deepest thanks to my advisor, Walter Cazzola, and Luca Favalli, a friend and companion on this adventure. Their unwavering patience and guidance have been invaluable in assisting me with this project, especially during the times when I was unfamiliar with the intricacies of language development and compilers. I would also like to take a moment of the reader's time to express my gratitude to my sister, Federica Bertolotti, who has consistently offered her assistance whenever I grappled with mathematical complexities.. Lastly, I am profoundly grateful to my family and friends who have patiently endured my moments of frustration and provided unwavering support. Their presence has been a constant source of strength and encouragement.

Before ending this preface, allow me to introduce you to the ★piler. The ★piler serves as a transpilation infrastructure, enabling the creation of both transpilers and compilers. This project emerged from the need to reuse existing language ecosystems. Often, when a new language gains popularity, developers expend significant effort recreating software libraries already available for established or older languages. As new languages emerge, the software community shifts focus, perhaps converging to an ultimate language or just endlessly following trends. Over the past year, I have contemplated how to simplify the introduction of new languages, ensuring comprehensive ecosystems are readily accessible. The ★piler represents my modest attempt to tackle this complex problem.

1 Prelude

In this hopefully short chapter, we will delve into the concepts necessary to understand the ★piler inner workings. We will start with a brief introduction describing the motivation of the work in Sect. 1.1. We will continue to the very syntactic core of the ★piler—languages (Sect. 1.2.1), grammars (Sect. 1.2.2), and parse trees (Sect. 1.2.3). Sect. 1.2.4 discusses how these objects are implemented in the ★piler. Similarly, Sect. 1.3 describes the necessary semantic objects (Semantic Domain Sect. 1.3.1, Semantic Context Sect. 1.3.2, and Denotational semantics in Sect. 1.3.3). Alongside the theoretical background, we also provide minimal code implementations in Sect. 1.3.4. Beyond syntax and semantics, we will introduce two more concepts: Metric Spaces (Sect. 1.5) and the A^* algorithm (Sect. 1.6). Certainly, readers can feel free to skip sections with which they are already acquainted.

1.1 Motivation

Before delving into the background theory, let's establish the motivation behind this work. As mentioned earlier, the development of compilers and transpilers is an intricate task. These endeavors involve numerous interconnected components that must function seamlessly together. Additionally, these tools exist in a state of perpetual evolution. Each year, new standards introduce fresh features to programming languages, sometimes even phasing out older or more convoluted elements. In some cases, language development is bifurcated to maintain multiple standards simultaneously, exemplified by the parallel existence of Python 3 and Python 2. However, it's not only the languages themselves that undergo continuous transformation; the entire language ecosystem is in a state of flux.

Consider the programming libraries that distill complex concepts into a few API calls. These libraries are in a constant state of development, and they can become so pivotal that they sway the choice of programming language itself. Yet, the language ecosystem encompasses more than just libraries. It includes interactive development environments (IDEs), debugging tools, software analysis utilities, and a multitude of other components.

Clearly, the amalgamation of all these components yields a dynamic yet intricate ecosystem in a constant state of evolution. Effectively addressing the complexity arising from these environments is of utmost importance because they constitute the bedrock of modern computation. Bugs or vulnerabilities within any of these components can

have far-reaching consequences, affecting thousands of devices and other software elements.

While some of these ecosystems may be completely independent, a significant amount of redundancy can be observed. Developers working with one programming language often find themselves addressing similar application domains as those working with another language. Consequently, similar libraries may emerge in different languages. Consider, for instance, the case of Python and Java, both of which provide libraries for logging purposes. While these libraries share a common objective, they are developed independently and offer slightly different APIs. For developers transitioning between these languages, adapting from one library to another, despite addressing the same issue, can prove to be a perplexing experience.

An even more challenging scenario arises when an application domain is easily manageable in one programming language but proves to be daunting in another. For instance, while one programming language might boast a comprehensive library for symbolic calculus, another may lack access to similar functionalities. As a result, developers are compelled to select one language over another primarily due to the mature ecosystem it offers, rather than the intrinsic features of the language itself. This predicament impedes the adoption of new programming languages that may lack a well-established ecosystem compared to their more established counterparts. Consequently, introducing innovative language features or entirely new programming paradigms becomes an arduous task.

Addressing such a complex issue presents a formidable challenge. However, we argue that transpilers offer a promising avenue for resolving this problem. With a couple of transpilers connecting two programming languages, it becomes possible to translate libraries from one language into the equivalent for the other programming language. This translation would enable developers from both language communities to leverage the same library. Simultaneously, the library could benefit from an expanded user base, allowing it to mature faster through the introduction of new features and bug fixes contributed by a wider audience.

Nevertheless, it is crucial to acknowledge that this ideal scenario may not always be attainable. Consider the stark contrast between programming paradigms, such as the functional and imperative paradigms. Transpiling a functional programming language into an imperative one might be a tractable challenge, but the reverse—transpiling an imperative language into a functional one—could prove exceptionally difficult. Furthermore, even if such a transpiler were developed, it could significantly alter the way developers interact with the library, potentially leading to substantial changes in the library's API. Additionally, transpilers are unidirectional, designed to convert from a source language to a target language. To achieve interoperability with a third programming language, developers would need to create additional transpilers and resources tailored to that specific language.

Yet, what are the alternatives?

- One option is to replicate the library from scratch in the target language. However, this approach typically demands a significant investment of time and resources.

Additionally, when the need arises for another library, starting from scratch each time can be inefficient and costly.

- Another approach is to utilize a foreign function interface (FFI) if the target language provides one for interoperating with the target library's source language. While this can be effective when possible, it relies on the availability of an FFI and still requires manual updates and maintenance.

- Virtual machines (VMs) offer an alternative avenue. In this scenario, a library could be directly accessible through the bytecode of the VM. This method can simplify access to libraries and facilitate cross-language usage when the languages share a common VM. However, if the goal is to achieve interoperability between languages supported by different VMs, additional challenges emerge, and alternative approaches must be considered.

Each of these solutions comes with its own set of advantages and drawbacks, and their applicability can vary depending on the specific scenarios and requirements. It's important to recognize that these solutions are not mutually exclusive. When addressing the challenges within the complex environment of language interoperability, different solutions may be needed depending on the specific cases.

In this work, our primary focus is on transpilers and their development. We introduce a framework for designing reusable transpilers, with the aim of minimizing the cost associated with introducing new languages to the transpilation pool.

1.2 Syntax

Indeed, the syntax of a programming language forms the foundation for understanding its implementation. The field of languages, grammars, and parsing is extensive, encompassing a wide range of concepts and techniques. Fortunately, for the purpose of implementing the ★piler, we will only require a basic understanding of these concepts and definitions. This will allow us to focus on the essential aspects needed to achieve our goals efficiently.

1.2.1 Language

A Formal Language, denoted as L, can be defined as a set formed by combining symbols from a given set Σ, commonly referred to as an alphabet. Formally,

Definition 1.2.1 (Language). *A language L over the alphabet Σ is a subset of Σ^*:*

$$L \subseteq \Sigma^*,$$

where Σ^* represents the set of all possible sentences over the alphabet Σ.

To illustrate this concept, consider the language $\alpha = \{a, aa, \dots\}$, which is formed by combining repetitions of the symbol a, and it belongs to the alphabet $\{a\}$. An example of finite language is the set $\{ab, abc, d\}$ built on the alphabet $\{a, b, c, d\}$.

Languages can be defined in various ways. Probably, the most natural methods are:

- Explicitly listing all the elements of the language, as we did previously with the language α.
- Another method for defining a language is through a characteristic function. The characteristic function $\mathcal{X} : \Sigma^* \to \{1, 0\}$ maps sentences belonging to the language to the value 1, while sentences not belonging to the language are assigned the value 0. For instance, the characteristic function for the language α can be defined as follows: 1 if every character c in the sentence w is equal to a, and 0 otherwise.

While these methods are straightforward, they are not always practical for defining languages.

A more practical approach to language definition is through grammars. In particular, we will focus on Context-Free Grammars (CFGs), which are commonly used to define most programming languages. However, it is important to note that CFGs are not capable of defining every possible language. The class of languages that can be defined by CFGs is known as Context-Free Languages.

1.2.2 Grammar

Grammars [20, 21] are incredibly powerful tools used to analyze and define languages from a syntactic standpoint. Intuitively, a grammar consists of four essential components. Firstly, we have the start symbol, which serves as the initial point of derivation. Starting from this symbol, we can apply rules that replace one or more symbols with a sequence of other symbols. Symbols can be either terminals or non-terminals. To formalize this concept, we express a grammar as follows:

Definition 1.2.2 (Grammar). *A grammar is a quadruple $G = (N, \Sigma, R, S)$. Where:*

- *N is the set of non-terminals.*
- *Σ is the set of terminal symbols.*
- *R is a set of production rules.*
- *S is a starting symbol such that ($S \in N$).*

Based on this definition, various types of grammars can be defined. Specifically, we are interested in Context-Free Grammars CFGs [79], which are widely used for language description. Other important classes of grammars are Regular (RGs) [35] and Context-Sensitive Grammars CSGs [52]. Each type of grammar has its own set of rules and restrictions, allowing for different levels of complexity and expressive power in defining languages [37]. In particular, CFG rules can only replace a single non-terminal symbol with any sequence of terminal or non-terminal symbols. Thus the definition can be reinstantiated as:

Definition 1.2.3 (CFG). *A CFG is a quadruple $G = (N, \Sigma, R, S)$. Where:*

- *N is the set of non-terminals.*
- *Σ is the set of terminal symbols.*
- *R is a set of rules. Where, $(A \to B) \in R$ iff $A \in N$, and B is a sequence of either terminals or non-terminals ($(N \cup \Sigma)^k$).*
- *S is a starting symbol such that ($S \in N$).*

As we mentioned, a grammar is used to define a language. But, we have seen two other methods that can be used to define languages. Therefore, it is only natural to want to convert one representation into the other. For example, given a particular grammar, we could ask ourselves what is the characteristic function representing the same language, or vice versa. We could ask ourselves if two representations are equivalent (meaning that they are capable of building the same class of languages). We could also ask what representations are more powerful (meaning that they are capable of building a class of languages that contains the other). We could easily lose ourselves in the hierarchy of languages [37] and their representations. However, this is a discussion that goes very far beyond the purpose of this manuscript. For the interested reader, these topics are discussed in [78, 54].

Instead, for our purpose, it is enough to know that CFGs are the most used grammars for modern programming languages. They cannot represent all languages but they are still extremely convenient grammars. Furthermore, it is very easy to check whether an element $w \in \Sigma^*$ belongs to the language defined as a CFG G. In general, we will denote with $\mathcal{L}(G)$ the set of sentences of the language defined by G:

Notation 1.2.1. *We will denote with $\mathcal{L}(G)$ the language defined by the grammar G.*

When we will need to say that a sentence belongs to the language defined by a certain grammar, we will use the symbol $\lhd$.

Notation 1.2.2. *Given a sentence $w \in \Sigma^*$, we will use the notation $w \lhd G$ as a short for:*

$$w \in \mathcal{L}(G)$$

A practical way to define a grammar (compared to Definition 1.2.2) is the Backus-Naur Form (BNF) notation [2] or its extended version (EBNF) [77]. In this formalism, the grammar is represented by a set of rules in the format `left ::= right`. In CFGs, the left-hand side of the rule consists of a single non-terminal symbol, while the right-hand side consists of a sequence of finitely many terminal or non-terminal symbols. Non-terminal symbols are enclosed in angular brackets (e.g. `<start>`). Terminal symbols are enclosed between double quotes (e.g. `"term"`). It is important to note that there should always be a rule with the starting non-terminal on the left-hand side. Now, allow me to provide an example of a simple grammar to illustrate this concept.

$$
\begin{aligned}
\langle \text{start} \rangle &::= \texttt{"("} \langle \text{start} \rangle \texttt{")"} \\
\langle \text{start} \rangle &::= \langle \text{start} \rangle \langle \text{start} \rangle \\
\langle \text{start} \rangle &::= \epsilon
\end{aligned}
$$

Grammar 1.1. *Balanced parentheses grammar.*

This grammar consists of three productions, all defined with a single non-terminal symbol, represented as `<start>`. In addition, there are two terminal symbols: `"("` and `")"`. The symbol ϵ denotes the null or empty symbol. The first rule specifies that the `<start>` symbol can be rewritten as the string `"("<start>")"`. This string can be further processed by applying any of the three rules. If we apply the last rule, we obtain the sentence $()$ since ϵ represents the null symbol. Notably, this grammar describes the language of balanced parentheses ($\{\epsilon, (), (()), (()(())), \dots\}$).

1.2.3 Parse Tree

Given a grammar G, we can define the language $\mathcal{L}(G)$. Now, suppose we have a sentence w, and we want to determine whether w belongs to the language ($w \lhd G$). One way to provide evidence for the question whether $w \in \mathcal{L}(G)$ is to present the sequence of derivations that lead from the start symbol to the sentence w. These derivations can be conveniently represented using a parse tree.

For instance, let's consider the previously discussed Grammar 1.1 for balanced parentheses and take the sentence $w = (())$. The corresponding parse tree for w would be as follows:

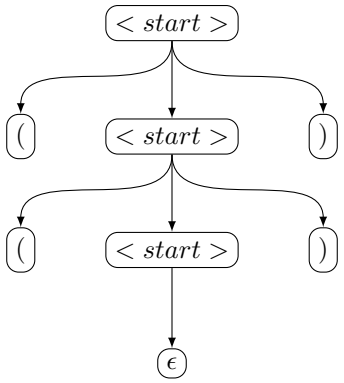


Figure 1.2. *Parse tree for the sentence $(())$ according to Grammar 1.1.*

The parse tree is typically the output of a parser (such as ANTLR [65]), which utilizes the grammar to determine the rules and non-terminals used to generate a given sentence [83].

In some cases, a sentence can be generated using different rules, resulting in multiple possible parse trees for the same sentence. When a grammar allows for multiple parse trees for a sentence, it is referred to as an ambiguous grammar. However, we will focus exclusively on non-ambiguous grammars in our discussion. Therefore, when we refer to a generic grammar, we imply a non-ambiguous CFG.

One important consequence of considering only non-ambiguous grammars is that the parse tree and the sentence itself are different representations of the same object. Hence, we will reuse the notation introduced in Notation 1.2.2 for parse trees. Consequently, we will state that the parse tree τ belongs to the language defined by the grammar G using the notation $\tau \in \mathcal{L}(G)$ or $\tau \lhd G$.

Given a parse tree τ, we may question whether it belongs to a specific grammar G. The process of verifying $\tau \lhd G$ is relatively straightforward. It entails verifying that each node, along with its children, adheres to one of the rules defined by the grammar G. Readers interested in exploring these topics can find a more comprehensive discussion in the work of Barthwal et al. [3].

1.2.4 Implementation

Languages, grammars, and parse trees play a crucial role in the development of complete programming languages. While these concepts have strong theoretical foundations, there are also numerous tools and libraries available that simplify their implementation with minimal code.

For instance, in the implementation of the ★piler, we utilized the popular [64, 92, 87] Python package called Lark.[1] Lark provides a powerful framework for working with CFGs by introducing a domain-specific language (DSL) for writing CFGs. With Lark, it is also possible to generate parsers based on the defined grammars.

Let us examine an example that demonstrates the implementation of Grammar 1.1 using the Lark library.

```
from lark import Lark
import rich
grammar = """
    start : "(" start ")"
          | start start
          | empty
    empty :
"""
parser = Lark(grammar, keep_all_tokens=True)
rich.print(parser.parse("(())"))
```

Listing 1.1. *Balanced parentheses grammar lark implementation.*

The provided Python code snippet showcases the definition of a grammar in the variable `grammar`. It then utilizes the Lark class to create a parser object named `parser`.

[1] `https://lark-parser.readthedocs.io`

This parser is applied to the string "*(())*" for parsing. The resulting output of this script can be observed in the following snippet (Snippet 1.2).

```
start
├── (
├── start
│   ├── (
│   ├── start
│   │   └── empty
│   └── )
└── )
```

Listing 1.2. *Output of the snippet 1.1.*

1.3 Semantics

Semantics plays a vital role in programming languages, as it provides the necessary rules and meaning to interpret and reason about programs. Without semantics, a programming language would be just a sequence of symbols that can be either accepted or not (Although it has its purposes even by itself).

There are several ways to specify the semantics, they can be classified into different types, such as operational semantics [67], denotational semantics [75, 76], and axiomatic semantics [94]. For our purposes, it suffices to focus on denotational semantics. Moreover, we only need a bare-bone understanding therefore, I will reduce the formalisms to the minimum necessary. The interested reader can complete these sections with material from [60]

1.3.1 Semantic Domain

In denotational semantics, the semantic domain $\mathcal{D}$ represents the mathematical objects that programs manipulate during execution. It establishes the range of values and computations within the language, allowing for the interpretation and evaluation of expressions, the execution of statements, and the determination of program behavior. By defining the semantic domain, a formal mapping is established between language constructs and mathematical objects, enabling precise and mathematical reasoning about program semantics. The semantic domain serves as a foundation for interpreting programs and facilitates the definition of denotations or interpretations for language constructs, specifying how they are evaluated and how they relate to the semantic domain.

Let us introduce a minimal programming language called "desk." The desk language, defined by Grammar 1.3, supports basic operations such as addition, variable assignments, and assignment concatenation. In desk, the operations are performed exclusively on integers, making the semantic domain $\mathcal{D}$ equivalent to the set of natural numbers, denoted as $\mathbb{N}$. The desk grammar definition is the following:

$$
\begin{aligned}
\langle \text{start} \rangle &::= \langle \text{asgn} \rangle \mid \langle \text{asgn} \rangle \texttt{";"} \langle \text{start} \rangle \\
\langle \text{asgn} \rangle &::= \langle \text{var} \rangle \texttt{"="} \langle \text{expr} \rangle \\
\langle \text{expr} \rangle &::= \langle \text{var} \rangle \mid \langle \text{int} \rangle \mid \langle \text{expr} \rangle \texttt{"+"} \langle \text{expr} \rangle \\
\langle \text{var} \rangle &::= \texttt{[a-z]+} \\
\langle \text{int} \rangle &::= \texttt{[0-9]+}
\end{aligned}
$$

Grammar 1.3. *Grammar for the desk language. For convenience, the last two rules define terminals using regular expressions.*

1.3.2 Semantic Context

The semantic context, also referred to as memory or store, represents the environment in which the language operates. It serves as a container for storing values from the semantic domain. A context is typically defined as a mapping between variable names $\mathcal{V}$ and the corresponding values in the semantic domain D, $\rho : \mathcal{V} \to \mathcal{D} \cup \{\bot\}$. Where the symbol $\bot$ represents a default empty value. The collection of all possible contexts is denoted by the letter P.

Next, it is convenient to define a way to update the semantic context, we will use the following notation:

Definition 1.3.1 (context update). *Given a context ρ, we denote with $\rho[x \leftarrow d]$ ($x \in \mathcal{V}$ and $d \in \mathcal{D} \cup \{\bot\}$) the updated semantic context, such that:*

$$\forall v \in \mathcal{V}, v \neq x : \rho[x \leftarrow d](v) = \rho(v) \wedge \tag{1.1}$$

$$\rho[x \leftarrow d](x) = d \tag{1.2}$$

1.3.3 Denotational Semantics

With the semantic domain and the semantic context set, we can now define the evaluation function. Denoted as $[\cdot](\cdot) : \mathcal{T} \times \mathrm{P} \to \mathcal{D} \cup \{\bot\}$, the evaluation function is responsible for mapping expressions with a given context to their corresponding values. It allows us to compute the result of expressions within the defined language.

In addition to the evaluation function, we will also utilize the execution function. Denoted as $[\![\cdot]\!](\cdot) : \mathcal{T} \times \mathrm{P} \to \mathrm{P}$, the execution function describes how statements operate on the context. It allows us to modify the context based on the execution of statements within the defined language.

A denotational semantic is typically defined by specifying both the evaluation and execution functions for all productions of a grammar. By providing these functions, we can determine the meaning and behavior of each program construct within the language. To illustrate this, let's define the semantics for the desk language (Grammar 1.3).

$$\forall a \in \text{<asgn>}, s \in \text{<start>}, \rho \in \mathrm{P} : [\![a; s]\!](\rho) = [\![s]\!]([\![a]\!](\rho)) \tag{1.3}$$
$$\forall v \in \text{<var>}, e \in \text{<expr>}, \rho \in \mathrm{P} : [\![v = e]\!](\rho) = \rho[var(v) \leftarrow [e](\rho)] \tag{1.4}$$
$$\forall e_0, e_1 \in \text{<expr>}, \rho \in \mathrm{P} : [e_1 + e_2](\rho) = [e_1](\rho) + [e_2](\rho) \tag{1.5}$$
$$\forall v \in \text{<var>}, \rho \in \mathrm{P} : [v](\rho) = \rho(var(v)) \tag{1.6}$$
$$\forall n \in \text{<int>}, \rho \in \mathrm{P} : [n](\rho) = int(n) \tag{1.7}$$

Here, the function *int* maps the syntactic values to their corresponding semantic domain, $\mathcal{D}$. Similarly, the *var* function maps the syntactic variables to their corresponding semantic domain, $\mathcal{V}$. In this example, Rule 1.3 describes the behavior of an assignment (`<asgn>`) concatenated (`";"`) with another desk program (`<start>`). This means that we need to execute the first assignment, obtain a new context, and then continue running the rest of the program on the new context. On the other hand, Rule 1.4 defines the behavior of a simple assignment. First, we evaluate the expression within the given context, and then we update the context by mapping the result of the evaluation to the specified variable name. To provide further clarification, let's consider an example execution. Consider the following desk program:

```
x = 1;
y = 2;
z = x + y
```

Listing 1.3. *A desk example program.*

The execution using the denotational semantics proceeds as follows:

$$\begin{aligned}
[\![x = 1; y = 2; z = x + y]\!](\rho_\perp) &= [\![y = 2; z = x + y]\!]([\![\text{x=1}]\!](\rho_\perp)) \\
&= [\![y = 2; z = x + y]\!](\rho_\perp[var(x) \leftarrow [1](\rho_\perp)]) \\
&= [\![y = 2; z = x + y]\!](\rho_\perp[var(x) \leftarrow int(1)]) \\
&= [\![y = 2; z = x + y]\!](\rho_1) \\
&= [\![z = x + y]\!]([\![\text{y=2}]\!](\rho_1)) \\
&= [\![z = x + y]\!](\rho_1[var(y) \leftarrow int(2)]) \\
&= [\![z = x + y]\!](\rho_2) \\
&= \rho_2[var(z) \leftarrow [x + y](\rho_2)] \\
&= \rho_2[var(z) \leftarrow [x](\rho_2) + [y](\rho_2)] \\
&= \rho_2[var(z) \leftarrow \rho_2(x) + \rho_2(y)]
\end{aligned}$$

We have abbreviated the full context representation with the symbols $\rho_\perp, \rho_1, \rho_2, \rho_3$:

- $\rho_\perp$ represents the empty context,
- $\rho_1 = \rho_\perp[var(x) \leftarrow int(1)]$,

– $\rho_2 = \rho_1[var(y) \leftarrow int(2)]$,

– while the explicit resulting context is $\rho_\perp[x \leftarrow 1][y \leftarrow 2][z \leftarrow 3]$.

1.3.4 Implementation

Just as syntactic elements are crucial in programming languages, semantic elements play an equally important role. Fortunately, there are numerous libraries available in the literature that allow us to apply theoretical concepts in practical implementations. One such example is the Lark Python package, which we used in the implementation of the ★piler examples. In addition to providing a DSL for defining CFGs, Lark also offers infrastructure for building interpreters and compilers.

To enhance our comprehension, let us consider an illustrative example. We will proceed with the implementation of the desk language using Lark. It is important to note that our implementation is minimalistic, lacking efficiency and completeness. Initially, we will define the grammar using Lark DSL. The chosen grammar aligns precisely with the one presented in Grammar 1.3.

```
grammar = """
    start : asgn | asgn ";" start
    asgn  : var "=" expr
    expr  : var | int | expr "+" expr
    var   : /[a-z]+/
    int   : /[0-9]+/
"""
```

Listing 1.4. *Lark implementation of CFG 1.3.*

Next, we will define the semantics for each production in the grammar. We will focus solely on the semantics of the assignment operation.

```
#...
class Asgn:
    def __init__(self, var, expr): self.var, self.expr = var, expr
    def exec(self, rho): return {**rho, self.var.name:self.expr.eval(rho)}
    def eval(self, rho): raise NotImplemented("Executable node only.")
#...
```

Listing 1.5. *Semantics for the assignment in Listing 1.4.*

Once the semantics are defined, we can utilize the Transformer class provided by the Lark library. This class allows us to traverse the parse tree in a bottom-up manner. By utilizing the Transformer, we can replace the syntactic nodes with executable ones. In the case of the assignment operation, we can implement it as shown in Listing 1.5. Once all the node in the parse tree have been replaced with executable nodes, we can execute the program `x=1;y=2;z=x+y`. Here, we show the resulting context.

```
class Desk(Transformer):
    def int(self, node): return Int(node[0].value)
    def var(self, node): return Var(node[0].value)
    def asgn(self, node): return Asgn(node[0], node[2])
    # ...
parser = Lark(grammar, keep_all_tokens=True)
tree = parser.parse("x=1;y=2;z=x+y")
Desk().transform(tree).exec(dict())
```

Listing 1.6. *Desk programming language implemenetation.*

```
{'x': 1, 'y': 2, 'z': 3}
```

Listing 1.7. *Output of the snippet 1.6.*

1.4 Search in Space

This section serves as an introduction to Metric Spaces and the A* search algorithm. Although these topics may appear unrelated to the preceding sections, they form the core foundation of the ★piler.

1.5 Metric Spaces

Metric spaces are fundamental mathematical structures that play a crucial role in various branches of mathematics, analysis, and applied sciences [18]. They provide a framework for studying the notions of distance, convergence, continuity, and many other fundamental concepts. The concept of a metric space extends the idea of distance from everyday Euclidean spaces to more general settings, allowing for the analysis of diverse mathematical objects.

Before formally defining a metric space, let us introduce the notion of *distance* function. A distance function is a function that assigns a non-negative value to pairs of elements in a given set, representing the "distance" or dissimilarity between them. It provides a formal way to quantify the separation or dissimilarity between objects or points in a space.

Definition 1.5.1 (distance function or metric)**.** *Given a set X, a distance function is $d : X \times X \to \mathbb{R}^+$ is a non-negative map such that:*

1. $\forall x \in X : d(x,x) = 0$.
2. *(positivity)* $\forall x,y \in X, x \neq y : d(x,y) > 0$.
3. *(symmetry)* $\forall x,y \in X : d(x,y) = d(y,x)$.
4. *(triangular inequality)* $\forall x,y,z \in X : d(x,z) \leq d(x,y) + d(y,z)$

With the notion of a distance function in mind, we can now formalize the concept of a metric space:

Definition 1.5.2 (metric space). *A metric space is a couple $\mathcal{M} = (X, d)$. Where X is a set, and d is a distance function on the set X.*

To gain a better understanding of these concepts, let's explore a few examples.

- One of the simplest metric spaces is the Euclidean space $(\mathbb{R}^n, d_e)$, where the distance function is defined as $d_e(x, y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$.
- However, the same set can also form a metric space using the Manhattan distance, denoted as $(\mathbb{R}^n, d_m)$, where $d_m(x, y) = \sum_{i=1}^{n} |x_i - y_i|$.
- Beyond Euclidean spaces, metric spaces can also be defined on sets of continuous and bounded functions. Let $\mathcal{C}[a, b]$ denote the set of continuous and bounded functions between the intervals a and b. In this case, the metric space can be represented as $(\mathcal{C}[a, b], d_c)$, where the distance function is defined as $d_c(f, g) = \sup |f(x) - g(x)| : a \leq x \leq b$.

1.6 A*

The A* search algorithm [34] is a widely used and highly efficient path finding algorithm in the field of artificial intelligence and graph theory [46, 100, 38, 62]. It is a popular choice for solving various optimization and search problems, especially in domains where finding the shortest path or optimal solution is crucial. A* combines the benefits of both breadth-first search and greedy best-first search by intelligently balancing the exploration of the search space using a heuristic function. This algorithm is particularly effective in domains with large or complex state spaces, as it systematically explores the most promising paths while keeping track of the estimated cost to reach the goal. With its ability to provide optimal solutions and its versatility in different problem domains, the A* search algorithm has become a fundamental tool for solving a wide range of real-world challenges.

As mentioned before, the A* algorithm requires an heuristic function. However, not any heuristic will do. To benefit from the optimal behavior, the heuristic needs to be non-overestimating. Overestimation happens when the heuristic estimate exceeds the actual cost of reaching a solution. To formalize this concept, let us begin with the definition of weighted graph and search problem:

Definition 1.6.1 (graph). *A graph is a couple $G = (V, E)$, where V is a set of nodes, and $E \subseteq V \times V$ is the set of edges.*

Definition 1.6.2 (weighted graph). *A weighted graph is a couple $G = (G' = (V, E), f)$, where G' is a graph, and $f : E \to \mathbb{R}$ is a non-negative map called weight function.*

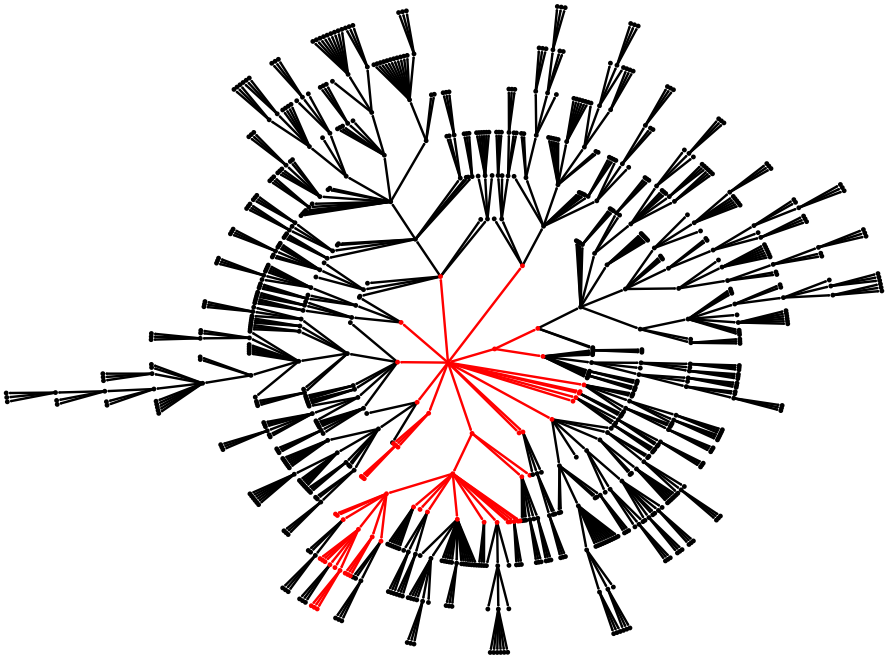


Figure 1.4. *Randomly generated graph embedded in a metric space. Nodes and edges explored by the A^* algorithm are shown in red.*

We provide an example graph in Fig. 1.5. Here, we defined a weighted graph with node set $V = \{0, 1, 2, 3, 4, 5\}$, edges $E = \{(0, 1), (0, 2), \ldots, (4, 5)\}$, and weight function $f((0, 1)) = 0.2, \ldots, f((4, 5)) = 9.99$. As shown in the example, weights can take any positive real value.

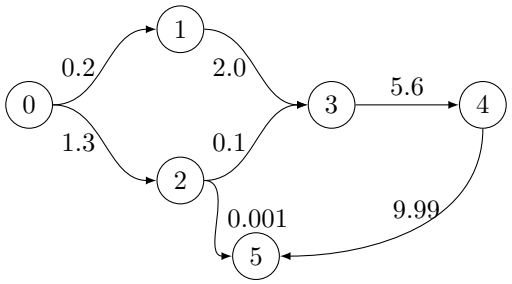


Figure 1.5. *An example of weighted graph.*

Next, we are ready to define the non-overestimating heuristic.

Definition 1.6.3 (non-overestimating heurisitc). *The heuristic $h : V \times V \to \mathbb{R}$ for the weighted graph $G = ((V, E), f)$ is non-overestimating iff $\forall a, b \in V : h(a, b) \leq h^*(a, b)$*

Here, h^* is the exact minimal distance required to go from the node a to the node b of G. Let us consider a few examples on top of Fig. 1.5. An always viable, but trivial heuristic is the zero-function as it cannot overestimate ($\forall a, b \in V : h_0(a, b) = 0 \leq h^*(a, b)$). A more useful heuristic, but rarely viable, is h^* itself ($\forall a, b \in V : h^*(a, b) \leq h^*(a, b)$). In general finding an heuristic that is fast to compute and non-overestimating is difficult. However, if we can prove that $G = ((V, E), f)$ is embedded in a metric space (V, d) such that, $\forall (a, b) \in E : f((a, b)) = d(a, b)$, then we can use d as heuristic function. We are guaranteed that h_d ($h_d(a, b) = d(a, b)$) does not overestimate by the triangular inequality. Let $h_d(a, b) = c$, c is the minimal cost of going from a to b if $(a, b) \in E$ otherwise, we need to traverse at least another node to reach b starting from a. Thus (by triangular inequality) $c \leq h^*(a, b)$. Let us formalize this last consideration:

Lemma 1.6.1 (non-overestimating heurisitc). *Let $G = ((V, E), f)$ be a weighted graph. If (V, d) is a metric space such that $\forall (a, b) \in V \times V : f((a, b)) = d(a, b)$ then d is a non-overestimating heuristic for G.*

Proof. Consider generic $a, b \in V$. We shall prove that $d(a, b) \leq h^*(a, b)$. First, suppose $(a, b) \in E$ then $d(a, b) = f((a, b)) = h^*(a, b)$. Now, suppose $(a, b) \notin E$ then to reach b from a, we need to traverse a third node, c. Thus, $h^*(a, b) \geq f((a, c)) + f((c, b)) = d(a, c) + d(c, b) \geq d(a, b)$. □

An example of graph embedded in a metric space is the road system with the Cartesian plane. Cities are nodes of the graph, edges are straight roads connecting one city to another, and the weight function of an edge is the length of the road. Of course to reach a city A from a city B you cannot take less time than using the straight road connecting A from B if one exists. Therefore, the Euclidean distance between cities is a non-overestimating heuristic for the city graph. For instance, let us examine Fig. 1.4, which displays a randomly generated graph. The red nodes indicate the nodes explored by the A* algorithm, commencing from the center and extending to the farthest external red node. This illustration vividly demonstrates that A* prioritizes exploring nodes that lead closer to the solution.

While a comprehensive commentary on the A* algorithm is beyond the scope of this work, for the sake of completeness, we also offer a concise Python implementation shown in Listing. 1.8.

```
def a_star_algorithm(source, target):
    openset  = set(source)
    closeset = set()
    g = {}
    g[source] = 0
    parents = {}
    parents[source] = source
    while len(openset) > 0:
        n = None
        for v in openset:
            if n == None or g[v] + h(v) < g[n] + h(n):
                n = v;
        if n == None  : raise ValueError("path does not exist")
        if n == target: return True
        for (m, weight) in n.children:
            if m not in openset and m not in closeset:
                openset.add(m)
                parents[m] = n
                g[m] = g[n] + weight
            else:
                if g[m] > g[n] + weight:
                    g[m] = g[n] + weight
                    parents[m] = n
                    if m in closeset:
                        closeset.remove(m)
                        openset.add(m)
        openset.remove(n)
        closeset.add(n)
    raise ValueError("path does not exist")
```

Listing 1.8. *A^* implementation in Python.*

2 ★piler

Now, we are poised to delve directly into the ★piler. We will commence with a concise overview in Sect. 2.1 to grasp the framework's entirety. Subsequently, we will progress to the theoretical framework in Sect. 2.2. Once we have established a firm understanding of the theory underpinning the ★piler, we will transition to the discussion of the implementation and its evaluation in Sect. 2.3. Finally, we will discuss the results in Sect. 2.4

2.1 Overview

While the concept behind the ★piler is relatively straightforward, the theoretical framework is rich in notation, making the theorems and lemmas quite intricate. Nonetheless, let us begin with the fundamental idea:

We want a transpiler that, given a set of transpilation units, a source program, a target language, composes the transpilation units to reach a program written in the target language. And, we want it fast.

To initiate our exploration, let us highlight the core essence: *We want a transpiler*. This encapsulates the crux of our pursuit—a program imbued with the capability to translate a source program from one language to a semantically equivalent rendition in a target language. A pivotal element within this endeavor is the concept of *transpilation units*—small functions designed to transform specific parts of code from the source language to the target language. Importantly, these units make these changes while preserving the semantics. It is worth noting that we do not assume anything about how these transpilation units are created; we are interested in their practical application. Naturally, the *source program* and *target language* are straightforward concepts and need no further introduction. When we say *composes the transpilation units*, we are describing the sequential application of these units until the transformation is complete—a bit like assembling a puzzle. This technique might sound familiar to experienced readers, as it involves repeatedly applying transpilation units until the source program becomes a program in the target language. However, the decisive distinction lies in our assertion that *we want it fast*. This aspiration for speed introduces complexity; achieving efficiency demands an intelligent deployment of transpilation units to prevent any waste of time.

Now, let's direct our attention to a key insight: employing a transpilation unit on a program is much like navigating the edges of a graph, where the nodes are the

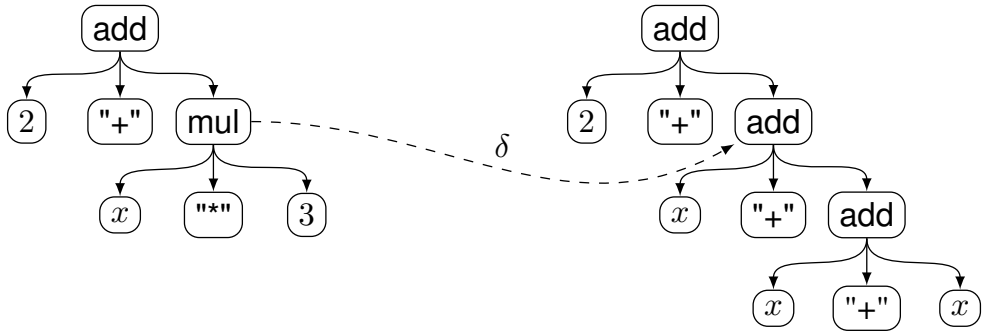


Figure 2.1. *The application of a delta function—δ—that transpiles the multiplication by a number to sequence of additions.*

different programs. This realization holds significance, for when we recognize that transpilation unit composition essentially is an exploration of an implicit graph, we can apply the techniques detailed in Sect. 1.4, making our efforts more manageable. Ultimately, the ★piler process can be seen as traversing a graph that emerges from the recursive application of these transpilation units. A significant portion of the theoretical framework is dedicated to embedding this implicit graph within a metric space. This allows us to leverage the efficiency of the A* search algorithm in our pursuit of finding a solution.

2.2 Theoretical Framework

2.2.1 Delta

Moving from the high-level framework to the theoretical intricacies, let us start with what we have previously termed as transpilation units. To reiterate, these units are simple functions designed to transform specific segments of a program from one language to another while preserving semantic equivalence. It is important to note that the realm of possibilities for a transpilation unit is wide open—there are no strict constraints (apart from maintaining semantic equivalence). They could range from complete transpilers to the partial transpilers that deal with the smallest language features, or even perform no transformation at all. They might even work on mixed languages. Clearly, the definition of a transpilation unit is flexible. Going forward, we'll refer to these transpilation units as deltas or δ for short:

Definition 2.2.1 (delta)**.** *Given $\tau \in \mathcal{T}$ (set of all possible parse trees). A delta is a function, $\delta : \mathcal{T} \to \mathcal{T}$ that maps a parse tree into another one such that:*

$$\forall \rho \in \mathrm{P}, \tau \in \mathcal{T} : [\![\tau]\!](\rho) = [\![\delta(\tau)]\!](\rho) \tag{2.1}$$

According to this definition, a delta is a function that maps a parse tree into another parse tree. Crucially, this transformation maintains the denotational semantics: regardless of all possible contexts $\rho \in \mathrm{R}$ and all potential parse trees $\tau \in \mathcal{T}$, executing τ with

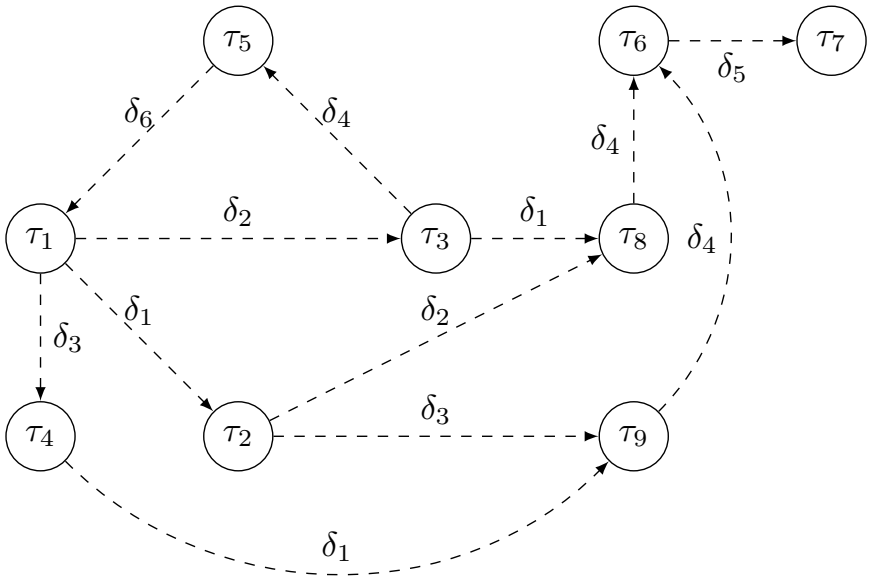


Figure 2.2. *A graph induced by the application of deltas $\{\delta_1, \delta_2, \delta_3, \delta_4, \delta_5, \delta_6\}$ on the syntax tree τ_1.*

context ρ yields the same result as executing $\delta(\tau)$ with the same context ρ. In other words, although τ and $\delta(\tau)$ may undertake different computational paths, their final results are always identical. As an example, consider Fig. 2.1. Here, the delta δ is used to transpile the multiplication expression from the left parse tree into a sequence of additions in the right parse tree. As one can see, regardless of the value of x (given by the context) the execution of the left tree and right tree is different but they achieve the same result.

One simple, yet crucial remark, is the fact that composing deltas leaves the semantics unchanged i.e., the composition of two deltas yields another delta:

Remark 2.2.1 (delta composition). *Given two deltas, δ_1 and δ_2, their composition, $\delta_1 \circ \delta_2$, is a delta.*

This remark can be verified by considering that $\forall \tau \in \mathcal{T}, \forall \rho \in \mathrm{P}[\![\delta_1(\delta_2(\tau))]\!](\rho) = [\![\delta_2(\tau)]\!](\rho) = [\![\tau]\!](\rho)$. Thus, any finite composition of deltas remains a delta. Thus, if the right delta are available, we can compose them to achieve a full language-to-language transpilation. In particular, a transpilation is defined as follows:

Definition 2.2.2 (transpilation). *Given a parse tree $\tau \in \mathcal{T}$. Given $\delta_{i_0}, \ldots, \delta_{i_n}$ deltas, we denote their composition $\delta_{i_0} \circ \cdots \circ \delta_{i_n}$ as $\vec{\delta}_I$, where $I = [i_0, \ldots, i_n]$. $\vec{\delta}_I$ performs a transpilation of τ to a grammar G iff*

$$\vec{\delta}_I(\tau) \lhd G$$

A transpilation for a parse tree τ to a grammar G, is nothing more than the composition of deltas $\delta_{i_0}, \ldots, \delta_{i_n}$ such that they achieve a parse tree from the target grammar G

when fed with the τ. As already mentioned previously, one can easily check when a tree τ belongs to the language defined by a grammar G ($\tau \lhd G$) [3].

A crucial observation to make here is that the application of deltas, initiated from a parse tree τ, leads to the creation of a graph. This concept is illustrated in Fig. 2.2. In this diagram, the initial parse tree is represented by τ_1. Upon applying deltas δ_1 and δ_2, two new parse trees, τ_2 and τ_3, are produced respectively. When starting from τ_3, the application of $\delta_4 \circ \delta_5$ results in returning to the initial tree τ_1. This arrangement highlights that within this framework, the transpilation search is equivalent to locating solution nodes within this induced graph. If we were not to care for speed, we could end the framework here as this would achieve our goal. Unfortunately, speed is one of those elements that cannot be ignored. Therefore, we continue our framework by trying to embed the induced graph into a metric space.

2.2.2 Search Graph

Let us introduce the search graph:

Definition 2.2.3 (search graph). *Let $\Delta = \{\delta_0, \ldots, \delta_N\}$ be a set of deltas. Let $V \subseteq \mathcal{T}_\Gamma$ be a set of parse trees from grammars $\Gamma = \{G_0, \ldots, G_M\}$. Let $E = \{(\tau_1, \tau_2) \in V \times V \mid \exists\, \delta \in \Delta \,.\, \delta(\tau_1) = \tau_2\}$. We call $S_{\Delta,\Gamma} = (V, E)$ the search graph.*

The search graph stands as a formal representation of our problem. Reference Fig. 2.2 once again. In this context, six delta functions $\Delta = \delta_1, \ldots, \delta_6$ were utilized. The iterative application of these deltas yielded a total of 7 parse trees, all originating from τ_1. Each of these parse trees pertains to a distinct grammar, denoted as $\Gamma = G_1, \ldots, G_7$. It's important to note that some G_i could equal G_j for certain $i, j \in 1, \ldots, 7$. Therefore, $S_{\Delta,\Gamma}$ visually encapsulates our search graph.

Now, we must introduce the objective of our search, which will define the search problem at hand:

Definition 2.2.4 (search problem). *The search problem is defined by the triple $(S_{\Delta,\Gamma} = (V, E), \tau, G)$, where $S_{\Delta,\Gamma}$ is a search graph, τ is a starting parse tree from V and G is a target grammar from Γ.*

The definition of the search problem essentially encapsulates our objective. It introduces the starting parse tree τ, which serves as the entry point for the search graph $S_{\Delta,\Gamma}$. It also presents the target grammar G, which indicates when, during the search, we have reached a solution. Consider again Fig. 2.2. In this context, the search problem could be denoted as $(S_{\Delta,\Gamma}, \tau_1, G_7)$, where $\Delta = \delta_1, \ldots, \delta_6$ and $\Gamma = G_1, \ldots, G_7$. It's evident that if $\tau_7 \lhd G_7$ and there are no other parse trees belonging to G_7 (i.e., $\nexists i \in 1, \ldots, 6 \,.\, \tau_i \lhd G_7$), then there exist multiple paths of deltas that lead to the solution. For example, the path $\vec{\delta} = [\delta_1, \delta_3, \delta_4, \delta_5]$ achieves the desired translation. It's important to note that even though there could be several paths solving the search problem, our interest lies in any one of them. Thus, we consider our problem solved if a solution exists, irrespective of the specific path taken to reach it.

Let us formalize the solution to the search problem:

Definition 2.2.5 (search solution). *Given the search problem $P = (S_{\Delta,\Gamma} = (V, E), \tau, G)$, a transpilation $\vec{\delta_I}$ is a solution for P when:*

1. $\vec{\delta_I}(\tau) \lhd G$.
2. $\forall i \geq 0 : \vec{\delta_{I[:i]}}(\tau) \in V$
3. $\forall i \geq 0 : (\vec{\delta_{I[:i]}}(\tau), \vec{\delta_{I[:i+1]}}(\tau)) \in E$

Let's examine the conditions that need to be satisfied for the transpilation $\vec{\delta_I}$ to be a solution for the search problem $(S_{\Delta,\Gamma} = (V, E), \tau, G)$:

1. $(\vec{\delta_I}(\tau) \lhd G)$: Naturally, the resulting parse tree from the transpilation (i.e., the application of $\vec{\delta_I}$) must belong to the language defined by the target grammar G.
2. $(\forall i \geq 0 : \vec{\delta_{I[:i]}}(\tau) \in V)$: Furthermore, we require that each of the parse trees traversed by $\vec{\delta_I}$ is actually a part of the search graph.
3. $(\forall i \geq 0 : (\vec{\delta_{I[:i]}}(\tau), \vec{\delta_{I[:i+1]}}(\tau)) \in E)$: Similarly, every edge traversed by $\vec{\delta_I}$ must be present within the search graph.

In essence, for $\vec{\delta_I}$ to be a valid solution, it needs to generate a parse tree that conforms to the target grammar, traverse valid nodes in the search graph, and follow valid edges within this graph.

The notation $I[: i]$ is a Python-like notation to take slices from a list defined as follows:

Notation 2.2.1 (slice). *Let $x = [x_0, \ldots, x_m]$, then $x[: k] = [x_0, \ldots, x_k]$ when $0 \leq k \leq m$. Otherwise, $x[: k] = x$ when $k > m$.*

The subsequent logical step is to establish a metric space for the search problem, thereby enabling the application of a heuristic for the A* algorithm. However, it is important to acknowledge that we currently lack knowledge of an existing metric that suits this context, nor have we been successful in formulating one. Nevertheless, there is still potential to make progress. By making slight adjustments to the definitions of the search graph, search problem, and search solution, we can introduce a modified version that opens up possibilities for the creation of an appropriate metric space. This adapted approach provides the necessary framework to incorporate a metric space that guides our transpilation searches. However, it is worth highlighting that the upcoming segment might be unnecessary under specific conditions, as outlined below:

- If a non-trivial heuristic function exists that can effectively guide the A* algorithm in the context of the search problem.
- Alternatively, if the search graph can be embedded in a metric space that successfully distinguishes between syntactically similar parse trees and dissimilar ones.

Before introducing the new versions of graph, problem, and solution, let us define a small utility function, referred to as node-set function with symbol λ:

Definition 2.2.6 (node-set). *Let us define the node-set function λ that given a parse tree $\tau = (V, E)$ returns the set of non-terminals from all non-leaf nodes of the tree, i.e.,*

$$\lambda(\tau) = \{v \,|\, v \in V \wedge \exists w \in V \;.\, (v, w) \in E\}\,.$$

This simple function is applied to parse trees and returns their non-leaf nodes. For example, the λ function applied on the root node of the left parse tree in Fig. 2.1 is $\lambda(\mathsf{root}) = \{\mathsf{add}, \mathsf{mul}\}$. Notice that the node-set function always returns a subset of the non-terminals of the grammar of the language used to write the program represented by the parse tree.

One important remark about transpilations and the node-set function is the following:

Remark 2.2.2. *If $\vec{\delta_I}$ is a transpilation from parse tree τ_1 to parse tree $\tau_2 \lhd G$ then:*

$$\lambda(\vec{\delta_I}(\tau_1)) \subseteq N_G$$

In this context, N_G represents the set of non-terminals within grammar G. This signifies that if a transpilation is oriented towards a particular grammar G, the resultant set of nodes must be confined within the collection of non-terminals present in G. If this observation were to be untrue, the transpilation would yield a parse tree featuring a non-leaf node that lies outside the set N_G of non-terminals. This would naturally lead to a contradiction since the transpilation aimed at G would then generate a parse tree τ such that $\tau \not\lhd G$.

2.2.3 Simplified Search Graph

Now, let us discuss the variation of the search graph/problem/solution respectively called *simplified search graph/problem/solution*, starting from the simplified search graph.

Definition 2.2.7 (simplified search graph). *Let $\Delta = \{\delta_0, \ldots, \delta_N\}$ be a set of deltas. Let $V \subseteq \mathcal{T}_\Gamma$ be a set of parse trees from grammars $\Gamma = \{G_0, \ldots, G_M\}$. Let $V' = \{\lambda(\tau) \mid \tau \in V\}$. Let $E' = \{(\lambda(\tau_1), \lambda(\tau_2)) \in V' \times V' \mid \exists\, \delta \in \Delta, \tau_1, \tau_2 \in V \;.\, \delta(\tau_1) = \tau_2\}$. We call $S'_{\Delta,\Gamma} = (V', E')$ the simplified search graph of the search graph $S_{\Delta,\Gamma} = (V, E)$.*

The formulation of the simplified search graph closely resembles that of the standard search graph, differing primarily in the definition of nodes. While the search graph nodes were individual parse trees and edges corresponded to deltas connecting parse trees, in this context, nodes represent node-sets from their respective parse trees. This adjustment could lead to different parse trees being consolidated under a single node. Consequently, the simplified search graph emerges as a *contraction*[1] (see Fig. 2.3) of the corresponding search graph. Within the simplified search graph, edges symbolize deltas that map node-sets from one parse tree to another node-set in a distinct parse tree.

Consequently, the definition of simplified search problem becomes:

[1] In this context, a contraction occur between two nodes u and v that are merged into a new node w.

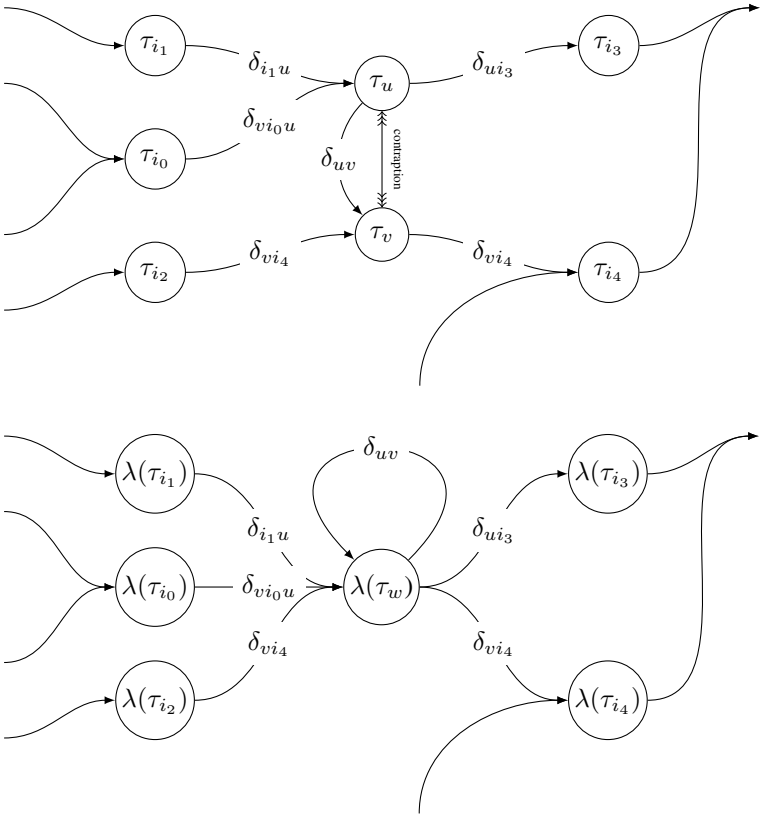


Figure 2.3. *A contraction between to two node representing parse trees with the same node-set. On top a portion of a search graph, below the same portion of the simplified search graph.*

Definition 2.2.8 (simplified search problem)**.** *The simplified search problem is defined by the triple* $(S'_{\Delta,\Gamma} = (V', E'), \tau, G)$*, where* $S'_{\Delta,\Gamma}$ *is a simplified search graph,* τ *is a starting parse tree from* V'*,* G *is a target grammar from* Γ*.*

The sole distinction in comparison to the search problem lies in the initial components of the triplet. Previously, it was based on a search graph, whereas now it relies on a simplified search graph. This subtle alteration results in a distinct interpretation of a solution, specifically dubbed as the simplified search solution:

Definition 2.2.9 (simplified search solution)**.** *Given the simplified search problem* $(S'_{\Delta,\Gamma} = (V', E'), \tau, G)$*, a transpilation* $\vec{\delta}_I$ *is a solution when:*

1. $\lambda(\vec{\delta}_I(\tau)) \subseteq N_G$.
2. $\forall i \geq 0 : \lambda(\vec{\delta}_{I[:i]}(\tau)) \in V'$
3. $\forall i \geq 0 : (\lambda(\vec{\delta}_{I[:i]}(\tau)),\ \lambda(\vec{\delta}_{I[:i+1]}(\tau))) \in E'$

In this scenario, for a transpilation to qualify as a solution, it only needs to map the initial parse tree τ to a node-set that falls within the non-terminals of the target

grammar G. This requirement is noticeably weaker. Consequently, a solution to the simplified search problem might not necessarily be a solution for the respective search problem. However, the converse remains true—a solution to the search problem also fulfills the requirements for the corresponding simplified search problem. As previously, conditions 2 and 3 of the theorem ensure that the transpilation $\vec{\delta}_I$ adheres to traversing nodes and edges that are part of the simplified search graph of the simplified search problem. This consideration yields to the first theorem:

Theorem 2.2.1. *The set of solutions for the search problem is a subset of the solution for the respective simplified search problem.*

Proof. The proof trivially follows by the fact that a solution for the search problem is always a solution for the simplified search problem. □

Suppose that we can quickly identify solutions for the simplified search problem. In that case, we can enumerate these solutions, check if they are also solutions for the original search problem, and if so, we have found a transpilation from τ to τ' where $\tau' \lhd G$. If they are not solutions (for the original search problem), we move on to the next solution for the simplified search problem. The problems arises when the number of solutions for the simplified search problem is not finite but there is no solution for the search problem. In this case, we may explore the search graph indefinitely. Consider this example, see Fig. 2.3, let τ_w be such that $\lambda(\tau_w) \in N_G$. Thus, $\vec{\delta}_{I_0} = [\ldots, \delta_{i_1 u}]$ is a solution for the simplified search problem. But also $\vec{\delta}_{I_1} = [\ldots, \delta_{i_1 u}, \delta_{uv}]$, $\vec{\delta}_{I_2} = [\ldots, \delta_{i_1 u}, \delta_{uv}, \delta_{uv}]$, ... are solutions. The previous, is an infinite countable set of solutions for the simplified search problem. If none of these are solutions for the original search problem, we would end up exploring them indefinitely.

Termination, may or may not be a deal breaker. On the other hand, if deltas are designed in such a way that it necessary to loop around applying the same deltas over and over to reach a transpilation, you have probably made purposely a bad design choice. Furthermore, as we will see in Sect. 3, we can address these issues by introducing further constraints on the deltas.

2.2.4 Metric Space

Now, we work towards building a metric space for the simplified search graph. In doing so, we firstly need to define a distance function between sets. We will use the distance discussed in [36]:

Definition 2.2.10 (set difference distance)**.** *Let $\mathcal{U}$ be a universe set, and let $A, B \subseteq \mathcal{U}$. The function $d_{sdd} : \mathcal{P}(\mathcal{U}) \times \mathcal{P}(\mathcal{U}) \to \mathbb{R}$ (where $\mathcal{P}$ represents the power set function) is defined as:*

$$d_{sdd}(A, B) = |A \cup B| - |A \cap B| \tag{2.2}$$

The set difference function (d_{sdd}) is a proper distance function [36]. A brief proof of this fact follows:

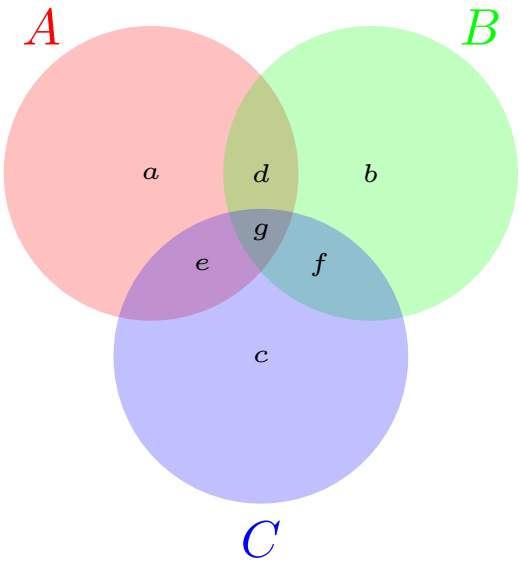


Figure 2.4. *Visual proof of the fact that* $|A \cup C - A \cap C| \leq |A \cup B - A \cap B| + |B \cup C - B \cap C|$. *Where* $a = A - B - C$, $b = B - A - C$, $c = C - B - A$, $d = A \cap B - C$, $e = A \cap C - B$, $f = B \cap C - A$, *and* $g = A \cap B \cap C$.

Proof. We need only to verify the 4 properties that distance functions must satisfy from Definition 1.5.1:

1. $\forall A \in \mathcal{P}(\mathcal{U}) \; : \; d_{sdd}(A, A) = |A \cup A| - |A \cap A| = 0.$
2. $\forall A, B \in \mathcal{P}(\mathcal{U}), A \neq B \; : \; d_{sdd}(A, B) = |A \cup B| - |A \cap B| \geq 1 > 0.$
3. $\forall A, B \in \mathcal{P}(\mathcal{U}) : \; d_{sdd}(A, B) = |A \cup B| - |A \cap B| = |B \cup A| - |B \cap A| = d_{sdd}(B, A)$.
4. $\forall A, B, C \in \mathcal{P}(\mathcal{U}) \; : \; d_{sdd}(A, C) \leq d_{sdd}(A, B) + d_{sdd}(B, C)$. We need to verify that $|A \cup C| - |A \cap C| \leq |A \cup B| - |A \cap B| + |B \cup C| - |B \cap C|$. Thus $|A \cup C - A \cap C| \leq |A \cup B - A \cap B| + |B \cup C - B \cap C|$. The rest of the proof can easily be seen by considering Fig. 2.4. The left side of the inequality is the cardinality of the set $a \cup d \cup c \cup f$. The right side of the inequality is the cardinality of set $a \cup e \cup b \cup f$ added with the cardinality of set $b \cup d \cup c \cup e$. As you can see, elements from the left side appear also on the right side of the inequality. Thus proving the triangular inequality.

Thus d_{sdd} is a proper distance function between sets. □

Now, this distance (d_{sdd}) is more than enough to build a metric space on top of the simplified search graph. However, we will introduce a novel distance function rooted on d_{rsdd}, denoted d^S_{rsdd}. d^S_{rsdd} introduces the solution set S, as the inputs of d^S_{rsdd} get closer to S the distance slowly collapse to a small value ($\frac{1}{2}$):

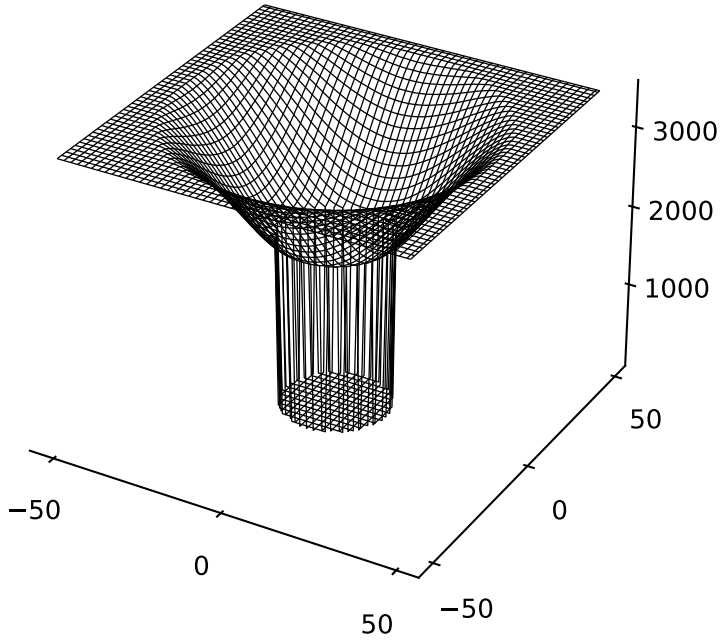


Figure 2.5. *Distance function d^S_{rsdd} graph. Each point (x, y) represents a set of a ball with center (x, y) and radius* 15*. The solution set is a ball with center in (0,0) and radius* 30*. As the set approaches the solution set, the distance decreases. When the set becomes a subset of the solution set, the distance becomes $\frac{1}{2}$.*

Definition 2.2.11 (relative set difference distance)**.** *Let $\mathcal{U}$ be a universe set. And let $A, B, S \subseteq \mathcal{U}$. The function $d^S_{rsdd} : \mathcal{P}(\mathcal{U}) \times \mathcal{P}(\mathcal{U}) \to \mathbb{R}$*

$$d^S_{rsdd}(A,B) = \begin{cases} d_{sdd}(A,B) & \text{if } A \not\subseteq S \wedge B \not\subseteq S, \\ d_{sdd}(S,B) & \text{if } A \subseteq S \wedge B \not\subseteq S, \\ d_{sdd}(A,S) & \text{if } A \not\subseteq S \wedge B \subseteq S, \\ \frac{1}{2} & \text{if } A \subseteq S \wedge B \subseteq S \wedge A \neq B, \\ 0 & \text{if } A \subseteq S \wedge B \subseteq S \wedge A = B \end{cases} \tag{2.3}$$

d_{rsdd} computes the distance between two sets relatively to a third set S.

The distance decreases as A and B get closer to S. Fig. 2.5 shows the behavior of $d^S_{rsdd}(A, S)$ wrt. the set S. The distance between A and S decreases when the set A approaches the set S. The distance collapses to the value $\frac{1}{2}$ when the set A is a subset of S but $A \neq B$. Any number $\in (0, 1)$ would comply with the distance function definition: we choose $\frac{1}{2}$ because it is the central number in the range.

Now, we have to prove that d^S_{rsdd} is truly a distance function as claimed:

Lemma 2.2.1. *d^S_{rsdd} is a distance function.*

Proof. Let $\mathcal{U}$ be the universe set. Let $\mathcal{X} \subseteq \mathcal{P}(\mathcal{U})$. Let $S \in \mathcal{X}$. Let us use $\overline{d}(\cdot,\cdot)$ instead of d^S_{rsdd} and $\underline{d}$ instead of d_{sdd}.

1. $A \in \mathcal{X} \implies \overline{d}(A,A) = 0$.
2. $A, B \in \mathcal{X} \wedge A \neq B \implies \overline{d}(A,B) > 0$.
3. $A, B \in \mathcal{X} \implies \overline{d}(A,B) = \overline{d}(B,A)$.
4. $A, B, C \in \mathcal{X} \implies \overline{d}(A,C) \leq \overline{d}(A,B) + \overline{d}(B,C)$.

The first three properties immediately follow from the definition of $\overline{d}$.The last property (triangle inequality) will be proved by exhaustively checking all cases.

Case $A = B = C$:

$$A, B, C \subseteq S: \qquad \underbrace{\overline{d}(A,C)}_{0} \leq \underbrace{\overline{d}(A,B)}_{0} + \underbrace{\overline{d}(B,C)}_{0}$$

$$A, B, C \not\subseteq S: \qquad \underbrace{\overline{d}(A,C)}_{\underline{d}(A,C)} \leq \underbrace{\overline{d}(A,B)}_{\underline{d}(A,B)} + \underbrace{\overline{d}(B,C)}_{\underline{d}(B,C)}$$

Case $A = B \neq C$:

$$\overline{d}(A,C) \leq \underbrace{\overline{d}(A,B)}_{0} + \overline{d}(B,C)$$

Case $A \neq B = C$:

$$\overline{d}(A,C) \leq \overline{d}(A,B) + \underbrace{\overline{d}(B,C)}_{0}$$

Case $A = C \neq B$:

$$\underbrace{\overline{d}(A,C)}_{0} \leq \overline{d}(A,B) + \overline{d}(B,C)$$

Case $A \neq C \neq B$:

$$
\begin{aligned}
A,B,C \subseteq S: &\quad \underbrace{\overline{d}(A,C)}_{1/2} \leq \underbrace{\overline{d}(A,B)}_{1/2} + \underbrace{\overline{d}(B,C)}_{1/2} \\
A \subseteq S \wedge B,C \not\subseteq S: &\quad \underbrace{\overline{d}(A,C)}_{\underline{d}(S,C)} \leq \underbrace{\overline{d}(A,B)}_{\underline{d}(S,B)} + \underbrace{\overline{d}(B,C)}_{\underline{d}(B,C)} \\
B \subseteq S \wedge A,C \not\subseteq S: &\quad \underbrace{\overline{d}(A,C)}_{\underline{d}(A,C)} \leq \underbrace{\overline{d}(A,B)}_{\underline{d}(A,S)} + \underbrace{\overline{d}(B,C)}_{\underline{d}(S,C)} \\
C \subseteq S \wedge A,B \not\subseteq S: &\quad \underbrace{\overline{d}(A,C)}_{\underline{d}(A,S)} \leq \underbrace{\overline{d}(A,B)}_{\underline{d}(A,B)} + \underbrace{\overline{d}(B,C)}_{\underline{d}(B,S)} \\
A,B \subseteq S \wedge C \not\subseteq S: &\quad \underbrace{\overline{d}(A,C)}_{\underline{d}(S,C)} \leq \underbrace{\overline{d}(A,B)}_{1/2} + \underbrace{\overline{d}(B,C)}_{\underline{d}(S,C)} \\
A,C \subseteq S \wedge B \not\subseteq S: &\quad \underbrace{\overline{d}(A,C)}_{1/2} \leq \underbrace{\overline{d}(A,B)}_{\underline{d}(S,B)} + \underbrace{\overline{d}(B,C)}_{\underline{d}(B,S)} \\
B,C \subseteq S \wedge A \not\subseteq S: &\quad \underbrace{\overline{d}(A,C)}_{\underline{d}(A,S)} \leq \underbrace{\overline{d}(A,B)}_{\underline{d}(A,S)} + \underbrace{\overline{d}(B,C)}_{1/2} \\
A,B,C \not\subseteq S: &\quad \underbrace{\overline{d}(A,C)}_{\underline{d}(A,C)} \leq \underbrace{\overline{d}(A,B)}_{\underline{d}(A,B)} + \underbrace{\overline{d}(B,C)}_{\underline{d}(B,C)}
\end{aligned}
$$

This concludes the proof: $\overline{d}(\cdot,\cdot)$ is a distance function. □

This result yields the second theorem:

Theorem 2.2.2. *Let $\mathcal{N}_\Gamma = \cup_{G\in\Gamma} N_G$ be the set of all non-terminals from grammars in Γ. Let $S \subseteq \mathcal{N}_\Gamma$. Then the couple $(\mathcal{P}(\mathcal{N}_\Gamma), d^S_{rsdd})$ is a metric space.*

Proof. The proof trivially follows from Lemma 2.2.1. □

This is the result we longed for in order to build a metric space on the simplified search problem. Recall that we introduced the distance function d^S_{rsdd} to prove the previous theorem; however, the same theorem holds for the distance function d_{sdd}. So, why did we initially opt for d^S_{rsdd} when we already had a simpler distance function? The reason for this choice is tied to the solution set. Remember, for the simplified search problem, each node representing a node-set contained in the set of non-terminals of the target grammar is a solution. If we used d_{sdd}, two different solution node-sets, let, say u and v, would have a distance greater than 0, even though both are solutions ($s_{sdd}(u,v) \geq 1$). Therefore, it is logical to construct a metric space in a way that ensures that all solutions are at distance 0 from each other, as they are equivalent from our perspective.

2.2.5 Heuristic

Now, we finally approach the last part of this section. In this last part our goal is to use the metric space previously introduced to build a heuristic function:

Definition 2.2.12 (heuristic). *Let $h_S : \mathcal{P}(\mathcal{N}_\Gamma) \to \mathbb{R}$ be the non-negative map*

$$h_S(A) = \begin{cases} 0 & \text{if } A \subseteq S \\ d^S_{rsdd}(S, A) & \text{otherwise} \end{cases} \tag{2.4}$$

Where $A, S \subseteq \mathcal{N}_\Gamma$

The function h_S is defined based on the solution set S. When a node, denoted as w, is considered a solution according to S, we set $h_S(w)$ to be 0. On the other hand, if w is not a solution according to S, we employ the distance function d^S_{rsdd} to gauge the estimated distance between w and S. However, we only require an estimate that does not overstate the actual distance. Let us proceed to prove this property:

Theorem 2.2.3. *h_S is an admissible heuristic (i.e., non-overestimating).*

Proof. Let $(S'_{\Delta,\Gamma}, \tau, G)$ be a simplified search problem. Where, we search for a transpilation starting from τ to a parse tree having non-terminals from N_G. Let $S = N_G$ and let τ^* be a parse tree such that $\lambda(\tau^*) \subseteq N_G$. We need to show that

$$h_S(\lambda(\tau)) \leq d^S_{rsdd}(\lambda(\tau), \lambda(\tau^*))$$

If $\lambda(\tau) \subseteq S$ then:

$$h_S(\lambda(\tau)) = 0 \leq d^S_{rsdd}(\lambda(\tau), \lambda(\tau^*))$$

As d^S_{rsdd} is a distance function, it cannot be negative. Otherwise,

$$\begin{aligned} h_S(\lambda(\tau)) &= d^S_{rsdd}(S, \lambda(\tau)) && (2.5)\\ &= d_{sdd}(S, \lambda(\tau)) && (2.6)\\ &\leq d_{sdd}(S, \lambda(\tau)) && (2.7)\\ &= d_{sdd}(\lambda(\tau), S) && (2.8)\\ &= d^S_{rsdd}(\lambda(\tau), \lambda(\tau^*)) && (2.9) \end{aligned}$$

□

All equations follows from either the definition of d^S_{rsdd} (Definition 2.2.11), or from the definition of the distance function. This theorem shows that, it is possible to apply A* to the simplified search problem. For each solution found by A*, we check whether it is a solution for the original search problem. We stop the search if it is a solution. Otherwise, we continue searching. Algorithm 1 shows the pseudocode for the transpilation. Algorithm 1 is a modified version of the A* algorithm to keep searching solutions. Notice that, both the search graph and the simplified search graph are never

Algorithm 1 A* adapted for the searching a transpilation

Require: Δ set of delta functions
Require: τ input parse tree
Require: G target grammar
  $C \leftarrow \{\}$
  $O \leftarrow \{\tau\}$
  $g \leftarrow$ map with default value $+\inf$
  $g(\tau) \leftarrow 0$
  $f \leftarrow$ map with default value $+\inf$
  $f(\lambda(\tau)) \leftarrow h_{N_G}(\lambda(\tau))$
  while $O \neq \varnothing$ **do**
    $C \leftarrow o \in O$ with lowest f value
    if $C \subseteq N_G \wedge c \lhd G$ **then return** c
    end if
    remove c from O.
    for all $\delta \in \Delta$ **do**
      score $\leftarrow g(\lambda(\delta(C))) + d^{N_G}_{rsdd}(\lambda(C), \lambda(\delta(C)))$
      if score $\leq \lambda(\delta(C))$ **then**
        $g(\lambda(\delta(C))) \leftarrow$ score
        $f(\lambda(\delta(C))) \leftarrow$ score $+ h_{N_G}(\lambda(\delta(C)))$
        if $\lambda(\delta(C)) \notin O$ **then**
          $O.add(\lambda(\delta(C)))$
        end if
      end if
    end for
  end while

explicitly built. The search graph is explored starting from τ on the fly by applying deltas from Δ.

In closing, it is worth highlighting that the algorithm presented here, a slight modification of A* (The modifications concern only the input, and the return condition), operates with a complexity of $\mathcal{O}(b^d)$. In this expression, b stands for the branching factor, and d represents the shortest path length from the initial node to the solution node. In our context, the branching factor is always bounded by $|\Delta|$, while d is bounded by $|V|$ (the number of nodes in the graph), resulting in a worst case complexity of $\mathcal{O}\left(|\Delta|^{|V|}\right)$. While we often have limited control over $|V|$ due to various problematic aspects, we can influence $|\Delta|$ in many cases. As evident, a smaller number of deltas leads to reduced time complexity.

With this algorithm, we can now conclude this section. However, this dissertation would be incomplete without at least one attempt to implement this framework. In the next section, we will delve into the practical implementation of the ★piler. We will evaluate its performance using straightforward programming languages. Additionally,

```
def int64 sort(int64* array, int64 len) does
    int64 i = 0;
    while i < len do
        int64 j = i;
        while j < len do
            if array[i] > array[j] do
                int64 tmp = array[i];
                array&[i] = array[j];
                array&[j] = tmp;
            done
            &j = j + 1;
        done
        &i = i + 1;
    done
    return 0;
done
```

Listing 2.1. *Bubblesort implementation with S programming language.*

we will compare the effectiveness of the A* search algorithm with a simpler Breadth First Search (BFS). This comparison will ultimately lead us to the conclusion that the utilization of A* is crucial for making such a framework applicable in real-world scenarios.

2.3 Concrete Framework

We can move beyond the theoretical framework to explore some practical aspects. We start by discussing the programming languages used to test the ★piler. In our development, we crafted three programming languages: S, S++, and S#. Each subsequent language in the sequence becomes progressively more complex. This set of languages, collectively referred to as the S family, or Simple languages, mirrors the evolution from C to C++ to C#. This section will commence with an exploration of S, the initial member of this language family.

2.3.1 The S programming language

The language S mimics a subset of C. It accounts for expressions, function definitions and declarations, while loop, if-then, and struct definitions. It supports native types: double, int64, int32, int8, void. For the S language, we also developed a compiler to LLVM [51] intermediate representation. Among the developed languages S is the only one that can be compiled directly to LLVM.

To better understand the capabilities of S, let us showcase a few examples starting from an implementation of the Bubblesort algorithm shown in Listing 2.1. This code snippet presents a basic implementation of the Bubblesort algorithm. It takes two inputs: a pointer to an int64 array (`array`) and an int64 value (`length`) indicating the

array's length (line 1). In lines 3-14, it employs loops indexed by `i` and `j` to traverse the array. When encountering a pair of unsorted elements (line 6), they are swapped (lines 7-9). Notably, both the while loop and the if-then statements exhibit syntax that is only slightly different from the C programming language. The distinction lies in the application of the `&` operator. Placed to the left of a variable, `&` returns the memory pointer to that variable. Conversely, `&[k]` designates the memory pointer of the k-th element in the array. The use of the `&` operator serves for pointer arithmetic and assignments. In S, the left side of assignments invariably comprises a memory pointer where the outcome of the right-side expression is stored.

The next example showcases the use of struct alongside a few other features:

```
1  struct X with
2      int64 x;
3      int64 y;
4      int64 z;
5  done
6  def int8* malloc(int64);
7  def void free(int8*);
8  def int64 start() does
9      X* x = &malloc(size of X) as X*;
10     &free(x as int8*);
11     return 1;
12 done
```

Listing 2.2. *Struct usage within the S programming language.*

Lines 1-5 define the struct `X`, composed of three int64 variables (`x`, `y`, and `z`). Lines 6 and 7 introduce two external functions, `malloc` and `free`, which are intended to be linked with the `libc` library for dynamic memory allocation. Finally, lines 8-12 present the `start` method. This specific signature signifies the entry point of any S program, akin to the main function in C. In S, functions are treated as variables, thus using the `&` operator on a function name yields the memory location of the function, which can be invoked using parentheses `(...)`. The `size of` operator functions analogously to the C `sizeof`, returning the memory size of the object to which it is applied (used in line 9). Meanwhile, the `as` operator serves for type casting, converting the type on the left side to the type specified on the right side (used in both lines 9 and 10). Thus, this small snippet, firstly dynamically allocates an `X` object (line 9), then it frees the memory allocated and returns 1.

The next S code snippet showcases the usage of the `import` to deal with multiple files:

```
1 // src/increment.s file /////////
2 def int64 increment(int64 x) does
3    return x + 1;
4 done
5 // src/start.s file ////////////////////////
6 from "src/increment.s" import increment as inc;
7 def int64 start() does
8     return &inc(0);
9 done
```

Listing 2.3. *`import` usage within the S programming language.*

Let us assume that lines 1-4 are saved in the file `src/increment.s`, and lines 5-9 are saved in the file `src/start.s`. In lines 2-3, a simple function is defined that takes an Int64 value and returns the value increased by one. Line 6 imports the `increment` function from the `src/increment.s` file and renames it to `inc`. The `start` function (the entry point) returns the result of incrementing the value 0 using the `inc` function.

The following code snippet showcases the usage of the `auto` keywords alongside the struct initialization:

```
1 struct X with
2     int64 x;
3     int64 y;
4     X* next;
5 done
6 def int64 start() does
7     auto x = X{x:0, y:1, next:0 as X*};
8     return x.y;
9 done
```

Listing 2.4. *`auto` usage within the S programming language.*

Lines 1-4 define a simple struct named `X`, which consists of two int64 values, `x` and `y`, and a pointer to another struct of type `X`. This allows for the creation of recursively defined types, similar to C. Notably, in line 7, the keyword `auto` is used to automatically infer the type on the right side of the assignment. Line 7 also demonstrates the initialization of the `X` struct. The fields `x` and `y` are initialized to 0 and 1 respectively, while the `next` field is initialized as `0 as X*`, which is akin to assigning the C `NULL` value to a pointer. In line 8, the notation `x.y` return the value stored in the field `y` of the variable `x` of type `X` (the retrieved value is 1). To retrieve the memory location a field one uses the `&.` notation. For example, `x&.y=1` would store in the field `y` of `x` the value 1.

Alongside struct initialization, it is also possible to directly initialize arrays:

```
1  def int64 start() does
2      int64* x = [1,2,3,4];
3      int64** y = [[1,2,3],[1,2]];
4      return 0;
5  done
```

Listing 2.5. *Global variables within the S programming language.*

Array initialization happens in a pythonic way. For example, consider Listing 2.5. It shows two arrays initializations. The first one containing four elements from 1 to 4 (line 2). The second one contains two arrays with 3 and 2 elements respectively (line 3). Thus the type of the first array, x, is a pointer to int64. Meanwhile, the type of the second array, y, is a pointer to pointers of int64 elements.

The last language feature that we showcase for the S language are global variables:

```
1  def auto x = 1;
2  def int64 start() does
3      return x;
4  done
```

Listing 2.6. *Global variables within the S programming language.*

In line 1, the `def` keyword is used to define a global variable named x, which is assigned the value 1. The variable type is automatically inferred with the keyword `auto`. The `start` method merely returns the value of this global variable.

The S programming language is entirely implemented using Python. The syntax of S is defined as a grammar using the open-source Lark package. For executing S programs, we developed an LLVM compiler using llvmlite[2], which offers Python bindings for the LLVM C++ compiler infrastructure API.

In theory, the ★piler could be used for compiling S directly to executable object code. However, this would require writing the deltas to directly translate S to object code, which goes beyond the scope of a simple demo. Moreover, such an approach would tie the execution of S to a specific processor architecture, hindering portability and reproducibility.

2.3.2 The S++ programming language

Here, we introduce the S++ language, which is an extension of the S language, resembling the C++ programming language. The primary addition in S++ compared to S is the inclusion of classes.

[2] `https://llvmlite.readthedocs.io`

```
1  class Pair with
2      def int64 a;
3      def int64 b;
4      def Pair* start(Pair* this, int64 a, int64 b) does
5          this&.a = a;
6          this&.b = b;
7          return this;
8      done
9  end
```

Listing 2.7. *Class within the S++ programming language.*

In S++, classes are created using the keyword `class`, followed by the class name (as seen in Listing 2.7 with `Pair`). The class body is enclosed between the `with` and `done` keywords. Class fields are defined similarly to global variables in S. In Listing 2.7, two int64 fields, `a` and `b`, are defined in lines 2-3. Lines 4-7 depict the class constructor. The constructor must be named `start`, receive an uninitialized object as an argument, and return a pointer to the newly created class object. The constructor role is to initialize the object received as an argument.

Given that S++ does not have a garbage collector, classes have another special method, namely the `end` method:

```
1  def void free(int8*);
2  def int8* malloc(int64);
3  class String with
4      def int8* string;
5      def String* start(String* this) does
6          this&.string = &malloc(64);
7          return this;
8      end
9      def void end(XY* this) does
10         &free(this.string as int8*);
11         return;
12     end
13 end
```

Listing 2.8. *Destructor within the S++ programming language.*

The purpose of the end method is to release any dynamically allocated memory associated with the class. For instance, in Listing 2.8, memory allocation for 64 bytes is performed in the constructor (lines 5-8). This memory will be automatically deallocated by the end method (lines 9-11) once the object is no longer in use. However, note that the end method is automatically called only when the related object is allocated on the stack. In cases where the object is allocated on the heap or another form of dynamic memory, it is necessary to directly call the end method to free the allocated memory. Failing to properly call the end method can lead to memory leaks.

As a result of the difference between stack and heap allocation, there are two ways to

construct and object:

```
1  class Point with
2      def double x;
3      def double y;
4      def Point* start(Point* this, double x, double y) does
5          this&.x = x; this&.y = y; return this;
6      end
7      def Point* set(Point* this, double x, double y) does
8          this&.x = x; this&.y = y; return this;
9      end
10 end
11 def int64 start() does
12     Point* point_heap  = new Point(0.0,0.0);
13     Point* point_stack = Point{x:1.0, y:1.0};
14     point_head.end();
15     return 1;
16 end
```

Listing 2.9. *Stack vs heap instantiation within the S++ programming language.*

Consider Listing 2.9. In this example, lines 1-10 define a simple Point class that manages two double numbers. This class does not have a custom destructor since it does not deal with dynamically allocated memory. The start function (line 11-16) demonstrates two ways to allocate and construct Point objects. In line 12, the `new` keyword is used. This notation allocates memory on the heap, which means that explicit calls to the destructor are necessary to properly deallocate the memory, as shown in line 14. On the other hand, line 13 showcases a notation similar to struct initialization in the S programming language. This notation allocates the Point object on the stack, eliminating the need for explicit calls to the destructor. Stack-allocated objects in S++ are automatically destructed when they go out of scope, ensuring memory cleanup.

In contrast to the S programming language, there is no dedicated compiler for the S++ programming language. To execute an S++ program, it needs to be translated to the S programming language first. Afterward, the translated S program can be compiled into LLVM code for execution. This multi-step process allows S++ programs to be executed by leveraging the existing S compiler and the LLVM compiler infrastructure.

2.3.3 The S# programming language

The final language we will consider is S#. This programming language extends the S++ programming language and mimics C#. The primary change introduced by S# is the incorporation of a garbage collector. This addition aims to eliminate the use of pointers, which have been removed from the language. However, for the sake of simplicity, S# is not multithreaded. It keeps track of allocated memory and only frees the memory explicitly by calling the garbage collector.

```
1  class Test{
2      fun (Test -> Test) __init__(this) {
3          return this;
4      }
5      fun (Test->void) f1(this) {
6          Integer integer = new Integer(1);
7          return;
8      }
9      fun (-> int64) __main__() {
10         Test test = new Test();
11         gccollect();
12         return 1;
13     }
14 }
```

Listing 2.10. *Main class in the S# programming language.*

Consider Listing 2.10. Firstly, note the main method `__main__`. This method is enclosed within the class `Test`. However, this signature serves to identify the entry point in the program. Next, observe that the type of the method is fully specified to the left of the method name. For example, method `f1` (lines 5-8) takes a single argument of type `Test` and returns nothing, resulting in the signature `fun (Test->void) f1`. Another special method name is `__init__`, which serves as the constructor. Unlike in S++, S# does not require a special signature for the destructor method as a garbage collector is provided. In Listing 2.10, the garbage collector is called at the end of the `__main__` method using the built-in `gccollect()` (line 11).

The S# language also directly supports double quoted strings, which were not directly supported in S and S++. The listing 2.11 showcases a simple String class that manages `int8[]` array representing a strings. The String class also offers few utilities such as clone, concat, and equals.

The notation for native strings is the conventional one, using double quotes (lines 13-15). In this case, the class `String` uses an `int8[]` field named `buffer`, which represents a one-dimensional array of `int8` elements. Unlike the C++ style notation, S# does not support multi-dimensional arrays. However, this limitation does not affect the capabilities of S# since multi-dimensional arrays can always be treated as single-dimensional arrays. Consider also the print method of Listing 2.11. In line 12, the `print` built-in is introduced. This function, prints an int8 array to standard output.

With respect to the previous programming languages (S and S++), S# also introduces the limited for loop.

```
1  for i from this.size {
2      this.buffer[i] = value;
3  }
```

Listing 2.12. *For loop within the S# programming language.*

```
class String {
    var int8[] buffer;
    fun (String, int8[]->String) __init__(this, buffer) {
        this.buffer = buffer;
        return this;
    }
    fun (String -> int64) size(this) {...}
    fun (String -> String) clone(this) {...}
    fun (String, String -> String) concat(this, other) {...}
    fun (String, String -> int64) equals(this, other) {...}
    fun (String -> String) print(this) {
        print(this.buffer);
        return this;
    }
    fun (->int64) __main__() {
        String string1 = new String("A");
        String string2 = new String("B");
        String string3 = new String("AB");
        return string1.concat(string2).equals(string3);
    }
}
```

Listing 2.11. *String class in the S# programming language.*

Language	Feature
Common	native types: long, int, char, double, float literals: string, array, rationals, naturals import from arithmetic operators: +, -, *, /, % logical operators: ==, !=, >=, <=, <, >, other operators: cast to, size of, indexing, enclosed expression, function call statements: if-then, while loop, return, return void, skip, statement expression, assignement, auto assignement, declaration assignement.
S	native types: pointers struct definition function definition global assignement declaration expressions: dereference, reference to
S++	native types: pointers class definition: fields, constructor, destructor, methods function definition global assignement declaration expressions: new, indexing, dereference, reference to
S#	class definition: fields, constructor, methods garbage collector expressions: new, indexing, method call statements: for loop

Table 2.1. *List of language features for S, S++, and S# languages. All languages share a set of common features defined in the* `Common` *entry. The other entries describe the additions wrt. the common features. Notice that even if different languages share the same language feature there may be either syntactic or subtle semantic differences. For example, S++ and S#* `new` *language features in principle are the same but have different implementations.*

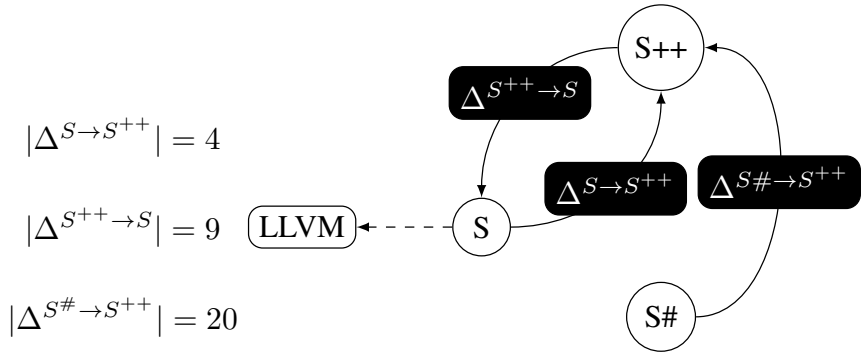


Figure 2.6. *The system presents three available languages. There are* 4 *deltas to translate S programs to S++ programs—denoted as* $\Delta^{S\to S++}$*. There are* 9 *deltas to translate S++ programs to S programs—denoted as* $\Delta^{S++\to S}$*. There are* 20 *deltas to translate S# programs to S++ programs—denoted as* $\Delta^{S\#\to S++}$*.*

The syntax for the limited for loop is showcased in Listing 2.12 (lines 1-3). The iteration variable is defined between the keywords `for` and `from`. The number of iterations is specified as an expression after the `from` keyword. In this case, during each iteration, the variable `i` takes values from 0 up to the result of evaluating `this.size - 1`. This construct is akin to traditional for loops in other programming languages and provides a way to iterate through a range of values within a limited scope.

Similar to S++, there is no direct compiler or interpreter for executing S# programs. However, it is possible to execute S# programs by translating them to S and then using the LLVM compiler to compile and subsequently execute the translated S# program. This process leverages the existing S language infrastructure for compilation and execution. Finally, we provide a small table summarizing the language features of S, S++ and S# in Table 2.1.

2.3.4 Translations

Now that we have dealt with the experimental programming languages (S, S++, and S#), We can move on to their translation. We developed a set of deltas necessary to translate S# to S++ ($\Delta^{S\#\to S++}$), S++ to S ($\Delta^{S++\to S}$), and S to S++ ($\Delta^{S\to S++}$). As a result, we can also translate S# to S using $\Delta^{S\#\to S} = \Delta^{S\#\to S++} \cup \Delta^{S++\to S}$. The resulting translations are summarized in Fig. 2.6. The same figure summarizes also the number of deltas used in each group: $|\Delta^{S\#\to S++}| = 20$, S++ to S $|\Delta^{S++\to S}| = 9$, and S to S++ $|\Delta^{S\to S++}| = 4$. For a total of 33 deltas.

Let us delve into the implementation of some of these deltas. Each delta is realized using a concept known as a "transformer," which is a fundamental component of the Lark parsing library. In Lark, a transformer is created by inheriting from the `Transformer` class. Within this inherited class, methods correspond to concrete syntax rules. For instance, `spplang_while` corresponds to the syntax rule governing the “while” loop feature. When a parse tree is processed, the `Transformer` parent class traverses it

and invokes the appropriate method for each and any encountered node. This invoked method processes the sub-tree rooted at that node, potentially altering it. The output of this method then substitutes the original sub-tree. In practice, a transformer could be employed to substitute every sub-tree governed by `spplang_while` with a single node, thereby transforming the structure.

Let us take a straightforward example. Our objective is to substitute instances of `spplang_int64` with `slang_int64`. It is important to note that the structural composition of these nodes remains unaltered, as their semantics is equivalent in both S++ and S.

```
from lark          import Token
from lark.tree     import Tree
from lark.visitors import Transformer

class Int64(Transformer):
    def spplang_int64(self, children):
        return Tree(Token("RULE", "slang_int64"), children)
```

Listing 2.13. *S to S++ delta for translating `spplang_int64` occurrences to `slang_int64` occurrences.*

In the provided code, we have created a subclass named `Int64` that inherits from the `Transformer` class. Within this subclass, we define a single rule, `spplang_int64`. This rule is designed to replace any occurrences of `spplang_int64` with a new `Tree` labeled as `slang_int64`. Notably, the structure of the children in the substituted tree remains unaltered.

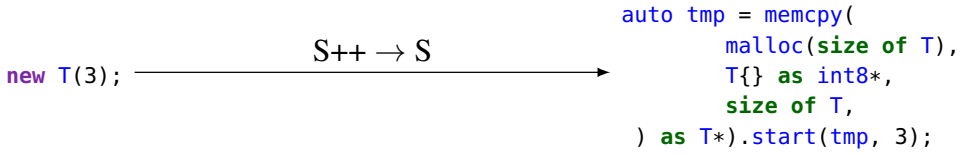
In our framework, each transformer corresponds to a delta. Given that our search algorithm complexity is bounded by $\mathcal{O}\left(|\Delta|^{|V|}\right)$, we may seek to manage the proliferation of deltas. This can be achieved by consolidating multiple rules into a single transformer instead of employing a separate transformer for each rule. It is important to emphasize that merging two transformers into a single one is always possible, which can effectively reduce the number of deltas if deemed necessary. This approach should be flexible and adaptable as needed.

Let us consider a new example:

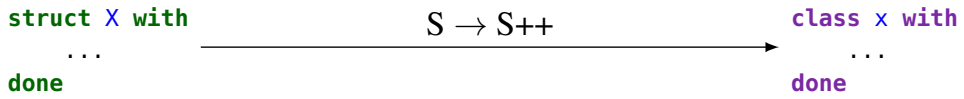
```
1  class News(Transformer):
3    def spplang_new(self, nodes):
4      return slang.parse(f'''
5        (
6          auto tmp = memcpy(
7            malloc(size of {get_type(nodes)}),
8            {get_type(nodes)}{{}} as int8*,
9            size of {get_type(nodes)}
10         ) as {get_type(nodes)}*
11       ).start(tmp {get_args(nodes)})'''
12     )
```

Listing 2.14. *S to S++ delta for translating `spplang_new`.*

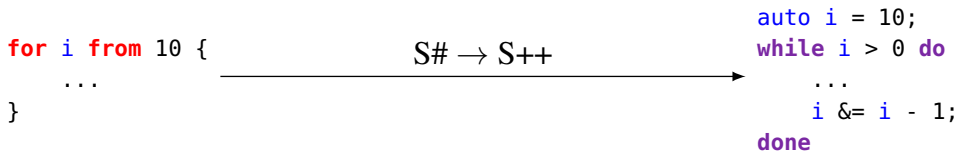
In this code section, we are transpiling the S++ `new` feature into a compliant S sub-tree.



(a) *S++ to S delta for new..*



(b) *S to S++ delta for structs..*



(c) *S# to S++ delta for for-loops .*

Figure 2.7. *Example of transpilations for different language features. Fig. 2.7a showcases the transpilation of the S++ `new` feature to the S programming language. Fig. 2.7b showcases the transpilation of the S `struct` language feature to the S++ programming language. Finally, Fig. 2.7c showcases the transpilation of the S# `for-loop` language feature to the S++ programming language.*

Suppose the instantiated type is `T`, and the constructor for `T` takes only one argument `3`. Thus, the result of `get_type(nodes)` is `T` and the result of `get_args(nodes)` is `3`. In lines 6-10, we utilize a temporary variable named `tmp`, which points to a portion of heap memory. This memory is initialized using the `memcpy` function. Recall that `memcpy` requires three arguments: a target memory pointer, a source memory pointer, and the size of the copy. In this context, the target memory pointer is allocated memory (line 7) of size `size of T`. The source memory pointer points to the stack-initialized object `T{} as int8*`. Remember that `T{}`, in S, signifies stack instantiation. The third argument, the size of the copy, is simply the size of the instantiated type `size of T`. Therefore, `tmp` now points to a newly instantiated object of type `T`, although it is not yet fully initialized. The constructor for `T` still needs to be called. We cast `tmp` to the appropriate type (i.e., `T*`, in line 10), and then invoke the constructor (i.e., the `start` method). Keep in mind that the `start` method always takes an instance of its own object. In this case, the argument is explicitly inserted in line 11 as `.start(tmp,...)`. Additional arguments for the `start` method are introduced using the `get_args(nodes)` function. Ultimately, this results in `tmp.start(tmp,3)`. Finally, we parse the resulting f-string into a tree which will replace the old one (line 4). Fig. 2.7a showcases an application of such a delta on a simple code snippet.

It is important to emphasize that the provided code snippet does not represent

the actual implementation. Incorporating a parsing step within a delta would make the delta computationally expensive. Additionally, numerous other intricacies need consideration. For instance, it is necessary to ensure that both `memcpy` and `malloc` are pre-defined. While these issues are resolvable, they complicate the discussion. To maintain clarity, we have chosen to offer a minimal code excerpt that offers an insight into how the delta might be implemented.

Now, let us see another example from the deltas that deal with S to S++ translations:

```
1  class Structs(Transformer):
2    def slang_struct(self, nodes):
3      #...
4      class_tree = Tree(Token('RULE', 'spplang_class'), [
5        Token('CLASS', 'class_'),
6        Tree(Token('RULE','spplang_identifier'), [
7          Token('__ANON__', get_struct_name(nodes))]),
8        Token('WITH', 'with_'),
9        Token('DONE', 'done')])
10     #...
```

Listing 2.15. *Delta to transpile S struct to S++ classes.*

Once again, we are presenting only a small portion of the code. As previously mentioned, the delta named `Structs` is responsible for replacing the sub-parse tree that defines S structs with equivalent sub-trees in S++. In this scenario, we have chosen to convert S structs into S++ classes. In lines 4-9, we are showcasing only a fragment of the resultant sub-tree. Here, we create an `spplang_class` node with an `spplang_identifier` labeled with the outcome of `get_struct_name(nodes)`. Subsequently, we will have to include all the struct fields and introduce a default constructor as well. Fig. 2.7b showcases an example of application on a minimal code snippet.

Let us move to the final delta that we will discuss:

```
1  class Fors(Transformer):
2    def ssharplang_for(self, nodes):
3      return [
4        Tree(Token('RULE', 'spplang_stmt_expr'), [
5          Tree(Token('RULE', 'spplang_auto_assignement'), [...]),
6          Token('DONE', 'done')]),
8        Tree(Token('RULE', 'spplang_while'), [
9          Token('WHILE', 'while'),
10         ...,
11         Token('DO', 'do'),
12         Tree(Token('RULE', 'spplang_block'), [...]),
13         Token('DONE', 'done')])
14     ]
```

Listing 2.16. *Delta to transpile S for loops to S++ while loops.*

This delta belongs to the set of transformations that deals with the translations

from S# to S++. Specifically, it handles the conversion of for-loop sub-trees into their corresponding while-loop counterparts in S++. This is achieved by substituting the for-loop construct with a while-loop structure. In lines 4-6, a variable is defined with the same name used by the iteration variable of the original for loop. Subsequently, in lines 8-13, a while loop sub-tree is constructed. This while loop utilizes the previously defined variable to replicate the iteration behavior of the original for loop, while maintaining the same loop body. An application of such a delta is shown in Fig. 2.7c.

2.3.5 Running Example

To better understand the overall framework, let us discuss a brief running example. Suppose we desire to transpile the following code snippet in S# to an S++ snippet:

```
1  class Point {
2    var double x;
3    var double y;
4    fun (Point,double,double->Point) __init__(this, x, y) {
5      this.x = x;
6      this.y = y;
7      return this;
8    }
9    fun (Point,Point->Point) add(this, other) {
10     this.x = this.x + other.x;
11     this.y = this.y + other.y;
12     return this;
13   }
14 }
```

Listing 2.17. *S# implementation of a class managing a 2-dimensional point.*

The provided code snippet introduces a basic class designed to handle points. The class contains fields `x` and `y`, representing the x and y coordinates respectively (lines 2-3). Additionally, the `Point` class offers a constructor for coordinate initialization (lines 4-8) and an `add` method for adding a given point to the current instance (lines 9-13). In order to translate listing 2.17, we need to address a few things:

- Replace the class definition syntax: Replace the open curly brackets with `with` and the closed curly bracket with `done`.
- Modify the syntax of defined fields (`x` and `y`): replace `var` with `def`.
- Adjust the syntax of defined methods (`__init__` and `add`): Replace `fun` with `def`. Replace the curly brackets with `does` and `done`. Move the type inside the parentheses.
- Rename the special method `__init__` to `start`, as constructors in S++ are defined using the name `start`.

- Address type differences: In S#, each defined object is implicitly a pointer to a heap memory location. Convert custom types to pointers (e.g., `Point` becomes `Point*`). Modify field access: when S++ expects a pointer, replace the dot notation with `&.` notation (e.g., `this.x` becomes `this&.x`).

- Consider that there is no overlap between non-terminals of S++ and non-terminals of S#. However, non-terminals such as identifiers have the same syntax. Thus, for a complete transpilation, we need to address such cases by simply renaming the non-terminals.

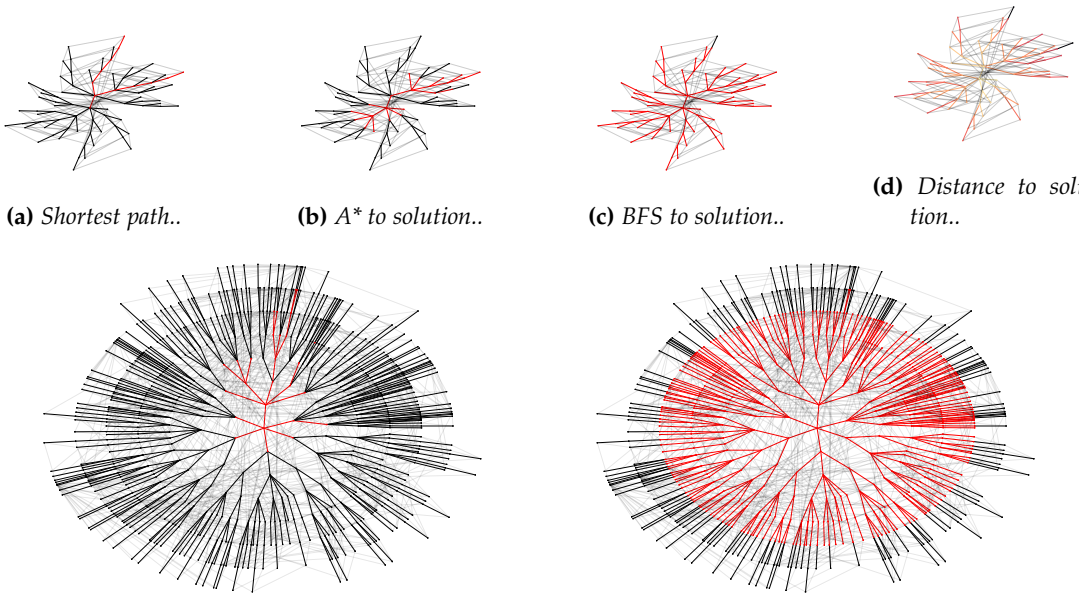
Additionally, in the provided code, the `new` keyword is not used. However, if we were to transpile an instantiation, we would also need to register the object in a specific structure. This registration process ensures that the object memory can be freed when the object is no longer in use and the garbage collector is called. This step is crucial for memory management in languages like S# without explicit pointer handling.

To accomplish these various small transpilations, we developed a set of specific deltas, each addressing a distinct aspect. This approach allows for a clear separation of concerns among the deltas and promotes a high degree of reusability. For instance, let us say we want to introduce S#-style for-loops into our new language. By including the desired syntax, we can readily utilize the existing deltas to translate the for-loop instances to both S and S++ syntax.

```
1  class Point with
2    def double x;
3    def double y;
4    def Point* start(Point* this, double x, double y) does
5      this&.x = x;
6      this&.y = y;
7      return this;
8    done
9    def Point* add(Point* this, Point* other) does
10     this&.x = this.x + other.x;
11     this&.y = this.y + other.y;
12     return this;
13   done
14 end
```

Listing 2.18. *S++ implementation of a class managing a 2-dimensional point.*

The transpilation starts with the parse tree representing Listing 2.17. Then, we apply all the available deltas. In this case, we have 20 deltas. Now, not all the deltas are applicable from the start, some delta depends on the execution of other deltas. However, after this step, we end up with at most 20 other parse trees. Next, among these parse trees, we need to choose the most promising one to explore. The idea is to choose the one that is closer to being a solution and that is not the result of a long transpilation (as we could end up stuck in a local minimum). This property is captured by the expected distance to reach the solution (given by the defined heuristic) with the traveled distance



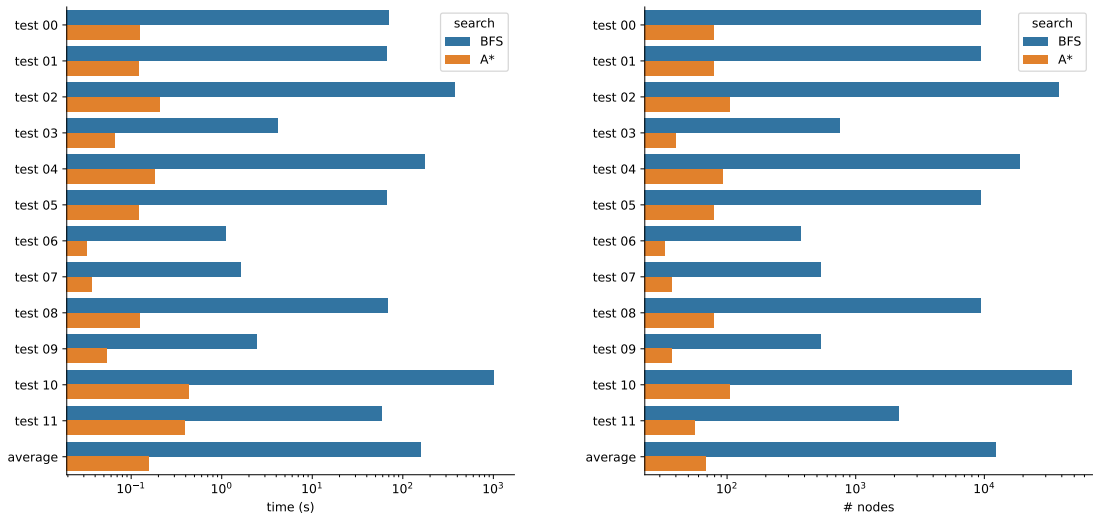
(a) *Shortest path..* **(b)** *A* to solution..* **(c)** *BFS to solution..* **(d)** *Distance to solution..*

(e) *A* to solution.* **(f)** *BFS to solution.*

Figure 2.8. *Graphs generated from the exploration of possible translations from a start S# program to the equivalent S++ program. The top row shows the full graphs induced using deltas from $\Delta^{S\#\to S++}$. The bottom row shows the graphs induced using a larger set of deltas $\Delta^{S\#\to S++} \cup \Delta^{S++\to S}$. Nodes are colored in black. Edges to already explored nodes are colored in light gray. Red colored nodes and edges show the explored paths of different algorithms.*

(given by the metric space). Having chosen the next parse tree, we apply again the whole battery of 20 deltas. Thus reaching a pool of at most 39 explorable parse trees. Again, we choose the most promising delta and we go on until we find a parse tree of which non-terminals are a subset of the solution set. Finally, we check if the found parse tree is a solution. If it is a solution, we end the search. Otherwise, if it is not a solution, we continue the search. The resulting translation is presented in Listing 2.18.

A delta application may depend on the application of previous deltas. For example, if we were to transpile the same `Point` class into the S language, we would need to transpile S# language features to the respective intermediate S++ language feature, as we did not develop any delta from S# to S (see Fig. 2.6). Of course, the path of deltas to perform the translation may not be known beforehand. Therefore, we need a search step to find the proper transpilation. As mentioned earlier, the application of deltas induces a graph on which we search for a solution. Consider Fig. 2.8, sub-figures from 2.8a to 2.8d shows the graph induced for translating the S# class `Point` into the S++ class `Point` using only the group $\Delta^{S\#\to S++}$. Figure 2.8a shows, in red, the shortest path from the root node to a solution. Figure 2.8b shows, in red, the nodes explored by the A* algorithm. Figure 2.8c shows, in red, the nodes explored using a BFS algorithm. Figure 2.8d shows, edges and nodes colored according to the distance from the solution wrt. the distance function d^S_{rsdd}. All these graphs, in light gray, show which edges



(a) *Test times.* **(b)** *Nodes visited..*

Figure 2.9. *Test times and nodes visited for each test in the S# → S++ translation task.*

connect a node to another explored node. Most noticeably, the A* algorithm explores a fraction of the nodes explored by the BFS algorithm. Now, let us consider Fig. 2.8e and Fig. 2.8f. These figures show the induced graph when using $\Delta^{S\#\to S++} \cup \Delta^{S++\to S}$. Most noticeably, the A* algorithm explores a number of nodes that is roughly the same with the previous case. Instead, the BFS algorithm explores a larger set of nodes, thus requiring more time.

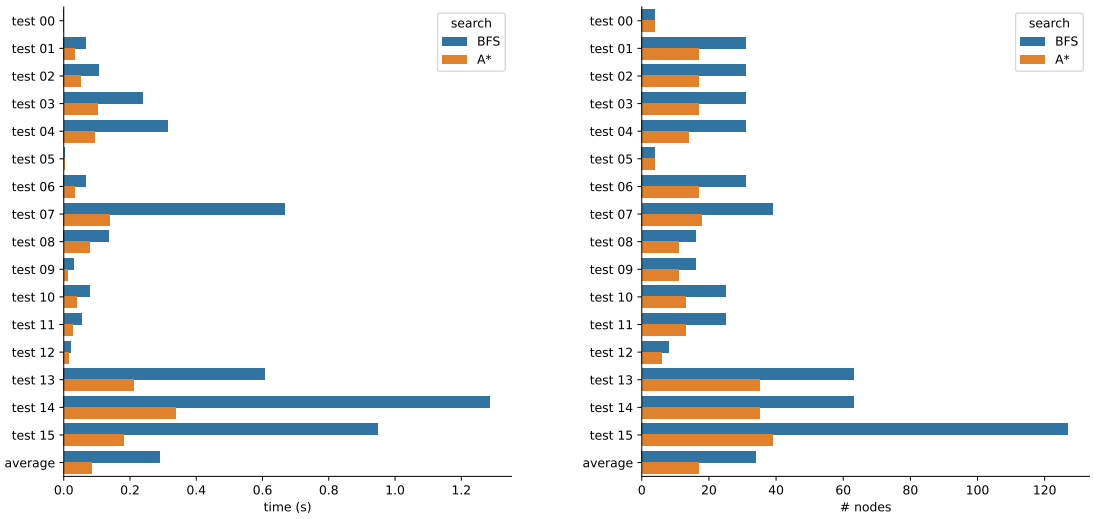
2.3.6 Evaluation

Having covered both the theoretical framework and the implementation details, we can now delve into the evaluation of the ★piler. It is essential to acknowledge that the ★piler is a compilation infrastructure that involves a search process, developed in Python. Consequently, in terms of speed, it might not match up to dedicated monolithic compilers. In this section, our focus will be on comparing the application of A* within the ★piler against a Breadth First Search approach.

The evaluation is performed on a PC with 32 GB of available memory and processor Intel Core i7-10700K.

We develop 12 tests for the S#→S++ translation task, 16 tests for the S++→S translation task, and 92 tests for the S→S++ translation task. The number of test for translate S→S++ is higher as we reused the test for the LLVM compiler. Each test is run for 5 times (for a total of 600 runs).

This translation task usually induces a large graph with thousands of nodes, as the number of deltas usable for the translation is 20. The main results are highlighted in Fig. 2.9. Figure 2.9a shows the transpilation times for all the 16 tests of the *S# → S++* translation task. The time axis is displayed in logarithmic scale. Using the A* search



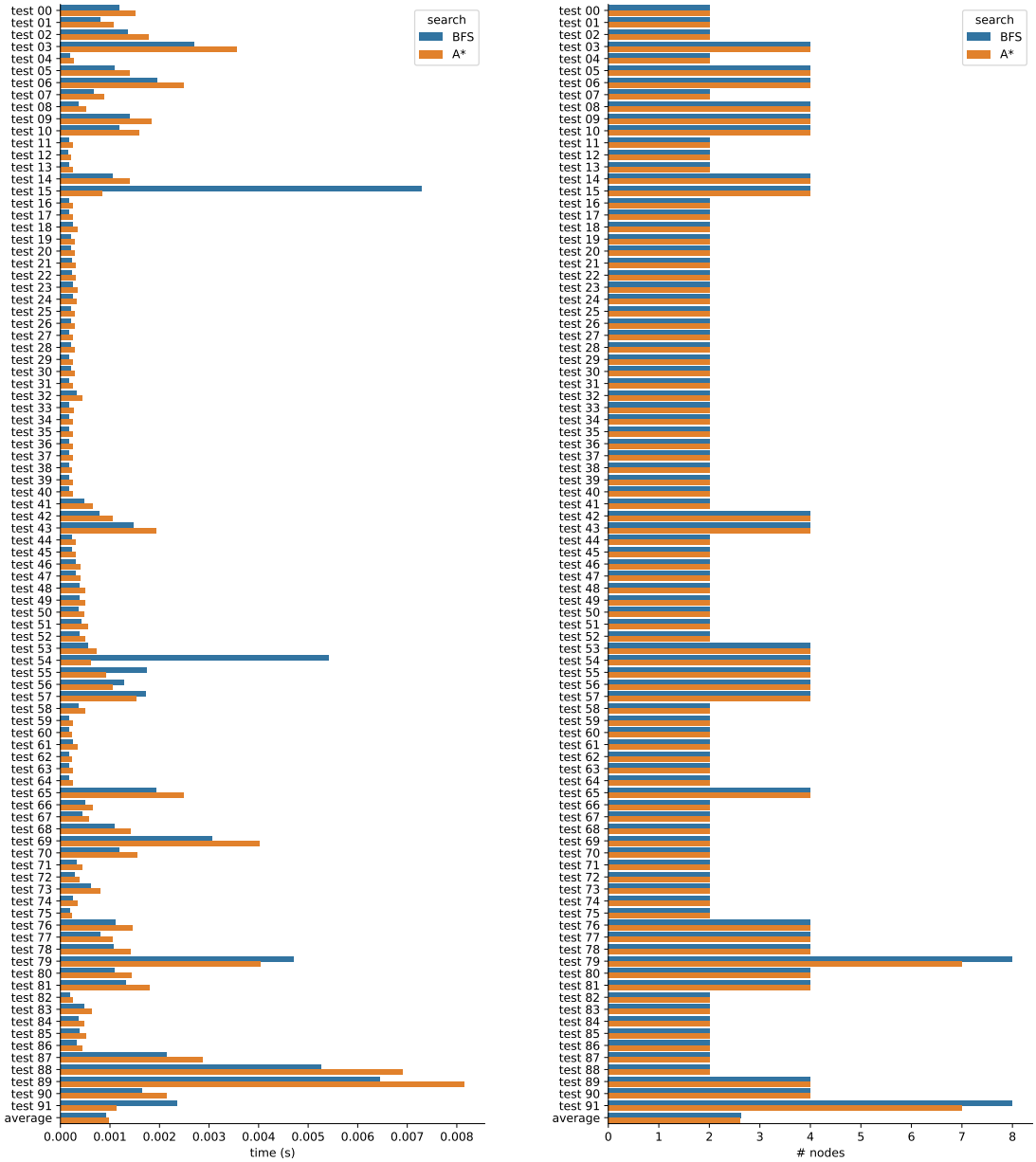
(a) *Test times.* **(b)** *Nodes visited.*

Figure 2.10. *Test times and nodes visited for each test in the S++ → S translation task.*

algorithm, to execute all the tests requires less than 1 s. On average, the completion time of all tests is 0.16 s. Instead, the tests (to translate S# to S++) may require several minutes to complete, ranging from 1 s to 20 min when using BFS. On average, the completion time is 158.52 s. Figure 2.9b shows the number of nodes visited for each test. On average, A* visits 68 nodes (ranging from 33 to 106) and BFS visits 12,265 nodes (ranging from 375 to 48,117). Thanks to the discussed heuristic in Sect. 2.2, A* is capable of exploring far fewer nodes compared to BFS, resulting in short translation times.

This translation task induces a far smaller graph with hundreds of nodes, as the number of deltas usable for translation is only 9. The main results are highlighted in Fig. 2.10. Figure 2.10a shows the transpilation times for all 12 tests of the $S++ \rightarrow S$ translation task. The time axis is displayed with a linear scale. Using the A* search algorithm, the average completion time is 0.09 s, ranging from 1 ms to 0.3 s. Instead, using BFS, the tests run for 0.29 s on average, ranging from 1 ms to 1.29s. Figure 2.10b shows the number of nodes visited for each test. On average, A* visits 16 nodes (ranging from 4 to 39) and BFS visits 34 nodes (ranging from 4 to 127). Again, the A* algorithm shows noticeable improvement over the BFS algorithm.

This translation task induces a very small graph with less than 10 nodes, as the number of deltas usable for translation is only 4. The main results are highlighted in Fig. 2.11. Figure 2.11a shows the transpilation times for all the 92 tests of the $S \rightarrow S++$ translation task. The time axis is displayed with a linear scale. Using the A* search algorithm, the average completion time is 0.966 ms, ranging from 0.2 ms to 0.9 ms. Using the BFS algorithm, the average completion time is 8.156 ms, ranging from 2 ms to 7 ms. Figure 2.11b shows the number of nodes visited for each test. On average, the A* algorithm visits 2.60 nodes, ranging from 2 to 7 nodes. Instead, the BFS algorithm visits



(a) *Test times.* **(b)** *Nodes visited.*

Figure 2.11. *Test times and nodes visited for each test in the S++ → S translation task.*

2.63 nodes on average, ranging from 2 to 8 nodes. In this scenario, the A* algorithm does not lead to improvements. Instead, due to the higher initialization costs, it leads to slightly lower performance.

2.4 Discussion

With this, we conclude our exploration of the ★piler. There are a few important considerations we would like to highlight. By now, we should possess a robust comprehension of the theoretical foundation of the ★piler, along with a reasonable grasp of its practical implementation. However, we have yet to delve into the factors that distinguish the ★piler from conventional monolithic compilers or language workbenches.

When comparing the ★piler to monolithic compilers, its trade-offs become evident. The ★piler prioritizes reusability and separation of concerns as inherent design principles. These features align with the essence of many language workbenches as well. However, the distinction that sets the ★piler apart lies in its unique approach. The ★piler goes beyond the conventional paradigms by merging elements from both language workbenches and search algorithms. While language workbenches provide an integrated environment for language creation and manipulation, the ★piler enhances this approach by incorporating heuristic-driven search algorithms. This combination addresses the challenges of transpilation with efficiency and accuracy. While the ★piler is not on par with modern language workbenches in terms of maturity or tooling offered, the ★piler can become the infrastructure onto which new language workbenches are built.

The strength of the ★piler originates from its ability to systematically navigate through vast language spaces, optimizing transpilation processes while maintaining compatibility between diverse languages. This capability stands out in comparison to traditional monolithic compilers and even many language workbenches.

In essence, the ★piler offers a novel transpilation strategy that leverages the strengths of both language workbenches and search algorithms. It bridges the gap between these approaches, providing a powerful tool that simplifies the complex task of language translation and compatibility.

Libraries migration Often, software libraries are specific for the language in which they are developed. Reusing these pieces of software becomes extremely difficult when different languages and different VMs are involved. However, by performing a translation, the same software developed in S++ can be reused with the S language. For example, we developed a small class managing strings in S++. The class uses stdio directives to efficiently compute string operations. By translating the S++ class, S developers can access a higher-level API to handle strings too. However, migration through transpilation requires a fair amount of carefulness. For example, in Python, class methods with specific names (such as `__init__` or `__getitem__`) can have a specific behavior. Instead these names add no special meaning for Java or C++. Therefore, if a Java method named `__getitem__` were to be transpiled in Python, another name would

have to be generated, and generating a meaningful new name may be difficult. However, when using the ★piler framework developers have the choice to use the deltas that fit the developer needs the most. For example, a developer may choose to use a delta that generates a new method name randomly, or he could use a delta that prompts for the new name when one is needed. Developers could use deltas that generated new method names using a deep learning architecture as [6, 5, 1]. Another difficult example is the Python `eval` and `exec` which allows to execute strings containing Python code. Since these strings cannot be inferred at compile time, when transpiling from Python to Java, the target language will need the ability (natively or with an external library) to evaluate/execute Python strings. The ★piler framework can handle these cases in two ways. 1) Developers employ a delta that translates the `exec` in a library call that can execute Python code strings. 2) Developers that do not have such a delta will not be able to translate Python source code using the `exec` builtin but they can still translate all the Python code that does not use the `exec` builtin.

Reusing compiler tests Compiler tests are one of the most valuable assets to verify the correctness of compilers. The research community has developed several tools to efficiently generate test suites for several compilers [19, 81]. However, generated suites are specific for the target language. Using the proposed framework, a test suite can be translated and reused to test new languages. A single suite may be capable to test different languages. For example, in our demonstration experiment, test programs developed for S++ and S# are also used to test the S language. Also, many of the S test programs are translated to S++ to test the language.

Reusing compiler components Introducing new features in existing languages can be difficult [7, 63, 57]. Instead, with ★piler the language developer only needs to add a syntactic construct and develop a delta performing the translation of the new feature. For example, in our demonstration experiment, we introduced the *for loop* in the S# language, then we added a delta function to translate the S# *for loop* to the S++ *while loop*. The same translation can be reused in other languages that implement the S# style for loop. For example, if we were to introduce S# style for loop to S++, we would need to only add the syntactic construct to the language and the system would automatically take care of the application of the respective delta to reach a translation.

Heuristic The proposed heuristic h_S is effective in exploring the language features inside a syntax tree to guide a translation system. However, there are cases when this heuristic is of little help. Recall that, the heuristic starts to lower when, during search, the current syntax tree shares language features with the solution set. Otherwise, the value of the heuristic does not shrink. For example, consider Fig. 2.5, on the edges of the graph the distance to the solution remains the same, thus not guiding the search towards the solution. In these cases, applying A^* is equivalent to applying a BFS. Meaning that A^* cannot help on those cases that require a chain of deltas that does not get close to the solution set early in the search. However, it is still possible to guide the

search, but it will require extra knowledge. For example, if we know that the translation needs to go through a certain set of language features S', we can craft another heuristic to guide the specific case:

$$h_{S,S'}(A) = \begin{cases} (0,0) & \text{if } A \subseteq S \\ (d^{S}_{rsdd}(S,A), d^{S'}_{rsdd}(S',A)) & \text{otherwise} \end{cases} \tag{2.10}$$

Among nodes that have the same distance to the solution set S, it promotes those that are close to the intermediate set S'. Of course, the heuristic can be extended with several intermediate sets to guide even delta paths that are expected to be extremely long.

Deltas development Deltas are functions developed in isolation from each other. This yields a system that can reuse already developed deltas to add language features to different languages easily. Yet during search, deltas can interact with each other unexpectedly as unforeseen situations appear. Overall, the deltas are the most crucial and difficult components in terms of development. They require additional care to ensure that it is applied as intended.

Deltas debugging Consider a transpilation path $\vec{\delta}$ that ends in an error (or an unexpected result). Most likely, this is due to a bug in one of the deltas used in $\vec{\delta}$. Yet, tracking the error is difficult, as it could have occurred at any point in the chain. Moreover, if we consider that these functions may be working on a very large syntax tree, pinpointing the bug may be extremely difficult and time-consuming. Therefore, if deltas are not organized properly, it may result in a brittle system.

Deltas as compilers We discussed a system that searches among a dataset of deltas for a path to a correct transpilation. However, developers may design deltas so that a specific chain works for every input program, *i.e.,* a compiler.

You can choose your transpilation Recall that the language S++ denotes the constructor method using the identifier `start`. If we were to translate a S# class that already contained a method named `start`, we would quickly run in a dilemma. We cannot transpile the S# `start` method as is, because it would be regarded as a constructor in the target language. Yet, changing its signature may be a problem as the S++ API would change. These cases may or may not require an *ad hoc* treatment. To solve these cases, one can choose whether to use the delta that handles methods with the name `start` or not. This feature does make for customizable transpilers that can handle a variety of cases depending on the developer needs. A variability managing framework applicable to the delta scenarios could be Feature Models [39, 73]. Further, approaches to self-optimization as [32] could be used to choose the best performing deltas depending on the environment.

Partial transpilers Transpiling a program in one language into another is not always possible. For example, transpiling a program written in a Turing-Complete language in a non-Turing-Complete language is not always possible. However, some programs may still be transpilable. Consider a program that performs only assignments and basic operations without relying on loops. Such a program can be transpiled into a non-Turing-Complete language (as long as it supports assignments). In these cases, the system will use only the deltas to translate assignments. If an unlimited loop is present in the source program, the transpilation will fail, as there will be no delta available to translate the loop into the target language.

3
Themes & Variations

In this concluding chapter, we will explore alternative design approaches to the ★piler. These proposed variations aim to enhance specific facets of the language development cycle, although they might introduce complexities in other areas. As we will observe, there will consistently be a delicate balance between ensuring system guarantees and the intricacies of implementation. It may happen that the result of a transpilation does not comply with the target grammar, in Sect. 3.1, we try to address these cases. In the ★piler, the search for the transpilation is done only when the parse tree to transpile is available, in Sect. 3.2, we try to address by searching ahead of time.

3.1 Variation 1

As a quick reminder, solving a transpilation with the ★piler often involves navigating through numerous search steps within the simplified search graph. In this context, we aim to restructure the system to eliminate the necessity for repeated searches. In the original ★piler version, the need for repeated searches stemmed from the uncertainty that even when the set of non-terminals aligned with the solution set (with the former being a subset of the latter), the located parse tree might not actually be a valid solution. Essentially, we needed to verify whether the identified parse tree conformed to the target grammar. In order to solve this issue, we need to reformulate the definition of delta as follows.

Definition 3.1.1 (augmented delta). *Let $G_1, \ldots, G_M$ be a set of grammars. Let N_i be the set of non-terminals of grammar G_i. An augmented delta, denoted δ^+, is a map $\delta^+ : \mathcal{T} \to \mathcal{T}$ such that:*

1. $\forall \rho \in \mathrm{P}, \tau \in \mathcal{T} : [\![\tau]\!](\rho) = [\![\delta^+(\tau)]\!](\rho)$
2. $\lambda(\delta^+(\tau)) \subseteq N_i \implies \delta^+(\tau) \lhd G_i$

Comparatively, the augmented delta is now linked to a specific set of grammars. For instance, when considering the transpilation between S, S++, and S#, these would represent the pertinent grammars. The conditions that a delta must satisfy have expanded to two: 1) The semantic equivalence of the resulting parse must still be maintained, akin to the previous deltas definition. 2) Additionally, when the set of nodes in the resulting parse tree ($\lambda(\delta^+(\tau))$) forms a subset of the non-terminals within

a grammar (N_i), then the resulting parse tree must adhere to that particular grammar ($\delta^+(\tau) \lhd G_i$).

Naturally, creating deltas that not only preserve semantic equivalence but also ensure the alignment of the resulting parse tree with potential target grammars is a more challenging task. Nevertheless, as we will observe, this requirement will ultimately lead to a more efficient framework.

As for Remark 2.2.1, note that the composition of augmented deltas remains an augmented delta:

Remark 3.1.1 (delta composition). *Given two augmented deltas, δ_1^+ and δ_2^+, their composition, $\delta_1^+ \circ \delta_2^+$, is an augmented delta.*

Proof. The proof follows from:

- Regarding the condition about semantic equivalence.

$$\forall \rho \in \mathrm{P}, \tau \in \mathcal{T} : [\![\delta_1^+(\delta_2^+(\tau))]\!](\rho) = [\![\delta_1^+(\tau)]\!](\rho) = [\![\tau]\!](\rho)$$

- Regarding the condition about grammar alignment. Let $\tau' = \delta_2^+(\tau)$

$$\lambda(\delta_1^+(\tau')) \subseteq N_i \implies \delta_1^+(\tau') \lhd G_i$$

□

As previously, any finite composition of augmented deltas remains an augmented delta. Thus, if the right deltas are available, we can compose them to achieve a full transpilation which we define as follows:

Definition 3.1.2 (transpilation). *Let $\tau \in \mathcal{T}$ be a parse tree. Given $\delta_{i_0}^+, \ldots, \delta_{i_n}^+$ augmented deltas, we denote their composition $\delta_{i_0}^+ \circ \cdots \circ \delta_{i_n}^+$ as $\vec{\delta_I}^+$, where $I = [i_0, \ldots, i_n]$. $\vec{\delta_I}^+$ performs a transpilation of τ to a grammar G iff*

$$\vec{\delta_I}^+(\tau) \lhd G$$

The definition of transpilation for this variant remains practically unchanged compared to the transpilation definition for the original ★piler (Definition 2.2.2). However, it is worth noting that here, the requirement $\vec{\delta_I}^+(\tau) \lhd G$ is equivalent to requiring $\lambda(\vec{\delta_I}^+(\tau)) \lhd N_G$, where N_G represents the set of non-terminals for grammar G.

Once more, we find ourselves in a similar scenario as before. The application of an augmented delta set to a parse tree results in numerous new parse trees to be examined. The repeated application of all the deltas to these unexplored parse trees opens a graph within which we search for a solution. As with the original approach, we must efficiently establish a heuristic to navigate this graph. Let us begin with the new definition of search graph:

Definition 3.1.3 (search graph). *Let $\Delta = \{\delta_0^+, \ldots, \delta_N^+\}$ be a set of augmented deltas. Let $V \subseteq \mathcal{T}_\Gamma$ be a set of parse trees from grammars $\Gamma = \{G_0, \ldots, G_M\}$. Let $E = \{(\tau_1, \tau_2) \in V \times V \mid \exists\, \delta \in \Delta \,.\, \delta(\tau_1) = \tau_2\}$. We call $S_{\Delta,\Gamma} = (V, E)$ the search graph.*

As before, we proceed by defining both the search problem and search solution. Both these definition are unchanged wrt. the original ★piler. However, note that the set of deltas, denoted as Δ, contains augmented deltas. Meanwhile, for the ★piler it contained standard deltas.

Definition 3.1.4 (search problem). *The search problem is defined by the triple $(S_{\Delta,\Gamma} = (V, E), \tau, G)$, where $S_{\Delta,\Gamma}$ is a search graph, τ is a starting parse tree from V and G is a target grammar from Γ.*

Definition 3.1.5 (search solution). *Given the search problem $(S_{\Delta,\Gamma} = (V, E), \tau, G)$, a transpilation $\vec{\delta}_I^+$ is a solution when:*

1. $\lambda(\vec{\delta}_I^+(\tau)) \subseteq N_G \;(\implies \vec{\delta}_I^+(\tau) \lhd G)$.
2. $\forall i \geq 0 : \vec{\delta}_{I[:i]}^+(\tau) \in V$.
3. $\forall i \geq 0 : (\vec{\delta}_{I[:i]}^+(\tau),\, \vec{\delta}_{I[:i+1]}^+(\tau)) \in E$.

In short, the search problem entails defining a search graph ($S_{\Delta,\Gamma}$), an initial parse tree (τ), and a target grammar (G). The search solution for this problem is a sequence of deltas ($\vec{\delta}_I^+$) that results in a parse tree conforming to the target grammar in terms of non-terminals ($\lambda(\vec{\delta}_I^+(\tau)) \subseteq N_G$). Notice that, such a requirement implies that the transpiled parse tree conforms entirely with the target grammar ($\vec{\delta}_I^+(\tau) \lhd G$). Moreover, this sequence of deltas ($\vec{\delta}_I^+$) must navigate through nodes and edges within the search graph (i.e., it cannot exit the search graph).

Now, instead of defining the simplified search/problem/solution, as we did for the ★piler, we directly embed the search graph within a metric space. Consider the following function:

Definition 3.1.6. *Let $\mathcal{T}$ be the set of all possible parse trees according to a defined set of grammars $G_1, \ldots, G_M$. Let S be a set of non-terminals $S \subseteq N_{G_1} \cup \cdots \cup N_{G_M}$. The non-negative map $\hat{d}_{rsdd}^S : \mathcal{T} \times \mathcal{T} \to \mathbb{R}$ is defined as:*

$$\hat{d}_{rsdd}^S(\tau_1, \tau_2) = d_{rsdd}^S(\lambda(\tau_1), \lambda(\tau_2)) + \frac{1}{2}(1 - \mathbb{1}[\tau_1, \tau_2])$$

Where $\mathbb{1}[\cdot, \cdot]$ is the indicator function such that $\mathbb{1}[x, y] = 1$ if $x = y$ and 0 otherwise.

We proceed by showing that $\hat{d}_{rsdd}^S$ is in fact a distance function wrt. the set of parse trees.

Lemma 3.1.1. *Let $\mathcal{T}$ be the set of all possible parse trees according to a defined set of grammars $G_1, \ldots, G_M$. Let S be a set of non-terminals $S \subseteq N_{G_1} \cup \cdots \cup N_{G_M}$. The non-negative map $\hat{d}_{rsdd}^S : \mathcal{T} \times \mathcal{T} \to \mathbb{R}$ is a distance function.*

Proof. In order to prove that $\hat{d}^S_{rsdd}$ is indeed a distance function, we need to show that all four requirements from Definition 1.5.1 hold.

1. Let $\tau \in \mathcal{T}$, we have

$$\hat{d}^S_{rsdd}(\tau, \tau) = d^S_{rsdd}(\lambda(\tau), \lambda(\tau)) + \frac{1}{2}(1 - \mathbb{1}[\tau, \tau]) = 0$$

2. Let $\tau_1, \tau_2 \in \mathcal{T}$ such that $\tau_1 \neq \tau_2$, we have

$$\hat{d}^S_{rsdd}(\tau_1, \tau_2) = d^S_{rsdd}(\lambda(\tau_1), \lambda(\tau_2)) + \frac{1}{2}(1 - \mathbb{1}[\tau_1, \tau_2]) \geq 0 + \frac{1}{2} > 0$$

3. Let $\tau_1, \tau_2 \in \mathcal{T}$, we have that

$$\begin{aligned}\hat{d}^S_{rsdd}(\tau_1, \tau_2) &= d^S_{rsdd}(\lambda(\tau_1), \lambda(\tau_2)) + \frac{1}{2}(1 - \mathbb{1}[\tau_1, \tau_2]) \\ &= d^S_{rsdd}(\lambda(\tau_2), \lambda(\tau_1)) + \frac{1}{2}(1 - \mathbb{1}[\tau_2, \tau_1]) \\ &= \hat{d}^S_{rsdd}(\tau_2, \tau_1)\end{aligned}$$

4. Let $\tau_1, \tau_2, \tau_3 \in \mathcal{T}$, we need to show that $\hat{d}^S_{rsdd}(\tau_1, \tau_3) \leq \hat{d}^S_{rsdd}(\tau_1, \tau_2) + \hat{d}^S_{rsdd}(\lambda(\tau_2), \lambda(\tau_3))$. That is:

$$\begin{aligned}d^S_{rsdd}(\lambda(\tau_1), \lambda(\tau_3)) + \frac{1}{2}(1 - \mathbb{1}[\tau_1, \tau_3]) \leq d^S_{rsdd}(\lambda(\tau_1), \lambda(\tau_2)) + \frac{1}{2}(1 - \mathbb{1}[\tau_1, \tau_2]) + \\ d^S_{rsdd}(\lambda(\tau_2), \lambda(\tau_3)) + \frac{1}{2}(1 - \mathbb{1}[\lambda(\tau_2), \lambda(\tau_3)])\end{aligned}$$

Now, we can reuse the fact that d^S_{rsdd} is a distance function, and thus $d^S_{rsdd}(\tau_1, \tau_3) \leq d^S_{rsdd}(\tau_1, \tau_2) + d^S_{rsdd}(\tau_2, \tau_3)$.

$$\begin{aligned}d^S_{rsdd}(\lambda(\tau_1), \lambda(\tau_3)) \leq& d^S_{rsdd}(\lambda(\tau_1), \lambda(\tau_3)) + d^S_{rsdd}(\lambda(\tau_2), \lambda(\tau_3)) \\ \leq& d^S_{rsdd}(\lambda(\tau_1), \lambda(\tau_2)) + d^S_{rsdd}(\lambda(\tau_2), \lambda(\tau_3)) + \\ & \frac{1}{2}(1 - \mathbb{1}[\tau_1, \tau_2]) + \frac{1}{2}(1 - \mathbb{1}[\tau_2, \tau_3]) - \frac{1}{2}(1 - \mathbb{1}[\tau_1, \tau_3])\end{aligned}$$

The last inequality follows from the fact that:

$$\frac{1}{2}(1 - \mathbb{1}[\tau_1, \tau_2]) + \frac{1}{2}(1 - \mathbb{1}[\tau_2, \tau_3]) - \frac{1}{2}(1 - \mathbb{1}[\tau_1, \tau_3]) \geq 0$$

This inequality can be simply verified by exhausting all cases regarding equality and inequality of τ_1, τ_2, and τ_3. The only case that would unverify the inequality is when $\tau_1 = \tau_2$ (so that the first term results in 0), $\tau_2 = \tau_3$ (so that the second term results in 0), and $\tau_1 \neq \tau_3$ (so that the third term results in $-\frac{1}{2}$). However, by transitive property of equality $\tau_1 = \tau_3$. Thus, this last case yields a contradiction.

Thus $\hat{d}^S_{rsdd}$ is a distance function. □

This lemma easily yields us to the following theorem:

Theorem 3.1.1. *Let $\mathcal{T}$ be the set of all possible parse trees according to a defined set of grammars $G_1, \ldots, G_M$. Let S be a set of non-terminals $S \subseteq N_{G_1} \cup \cdots \cup N_{G_M}$. Then $(\mathcal{T}, \hat{d}^S_{rsdd})$ is a metric space.*

Proof. The proof trivially follows from Lemma 3.1.1. □

Having established a proper metric space for parse trees, we can now define a heuristic based on this metric space. However, we must exercise more caution than in the original ★piler design. Unlike before, we cannot directly employ the distance metric as a heuristic function. In the prior design, the distance function operated on sets of non-terminals, and we were aware of the solution set. This time, the distance function pertains to parse trees, a distinct entity from the solution set. Consequently, directly measuring the distance between them is not feasible. To create a non-overestimating heuristic, we will need to return to employing d^S_{rsdd}.

Definition 3.1.7. *Let $\hat{h}_S : \mathcal{T} \to \mathbb{R}$ be the map defined as:*

$$\hat{h}_S(\tau) = \begin{cases} 0 & \text{if } \lambda(\tau) \subseteq S \\ d^S_{rsdd}(S, \lambda(\tau)) & \text{otherwise} \end{cases} \tag{3.1}$$

We proceed by showing that $\hat{h}_S$ is indeed a non-overestimating heuristic function. This is the subject of the next theorem.

Theorem 3.1.2. *$\hat{h}_S$ is a non-overestimating heuristic.*

Proof. In order to show that $\hat{h}_S$ is a non-overestimating heuristic, we need to show that, given a solution parse tree τ^* (recall that by definition of the solution set, $\lambda(\tau^*) \subseteq S$) and an arbitrary parse tree τ:

$$\hat{h}_S(\tau) \leq \hat{d}^S_{rsdd}(\tau, \tau^*)$$

i.e., $\hat{h}_S$ does not overestimate the actual distance to the solution. Notice that, the actual distance may depend on the specific topology of the graph. However, $\hat{d}^S_{rsdd}(\tau, \tau^*)$ is surely a lower bound for the true distance. If this were not the case, we would be breaking the triangular inequality of the previous Lemma, which would lead to an contradiction.

The rest of the proof follows by considering cases exhaustively:

- Case ($\lambda(\tau) \subseteq S$).

$$\begin{aligned} \hat{h}_S(\tau) &\leq \hat{d}^S_{rsdd}(\tau, \tau^*) \\ 0 &\leq \hat{d}^S_{rsdd}(\tau, \tau^*) \end{aligned}$$

  This is surely the case, as we showed that $\hat{d}^S_{rsdd}$ is a distance function, and as such, it cannot be negative.

– Case ($\lambda(\tau) \not\subseteq S$).

$$\begin{aligned}
\hat{h}_S(\tau) - \hat{d}^S_{rsdd}(\tau, \tau^*) &\leq 0 \\
d^S_{rsdd}(S, \lambda(\tau)) - d^S_{rsdd}(\lambda(\tau), \lambda(\tau^*)) - \frac{1}{2}(1 - \mathbb{1}[\tau, \tau^*]) &\leq 0 \\
d_{sdd}(S, \lambda(\tau)) - d_{sdd}(\lambda(\tau), S) - \frac{1}{2}(1 - \mathbb{1}[\tau, \tau^*]) &\leq 0 \\
-\frac{1}{2}(1 - \mathbb{1}[\tau, \tau^*]) &\leq 0
\end{aligned}$$

Which is definitely true.

□

In the initial version of the ★piler, the condition $\lambda(\tau) \subset N_G$ for a target grammar G did not automatically ensure that $\vec{\delta}(\tau) \lhd G$. Yet, with the introduction of augmented deltas, this has changed ($\vec{\delta}^+(\tau) \lhd G$). The consequence of this slight adjustment is a framework that eliminates the need to restart searches. This attribute is certainly advantageous for the framework, but it comes with a trade-off. The augmented delta approach imposes additional demands on the deltas, making their development more complex and intricate.

As we proceed to the following sections, we will observe that we can impose additional burdens on the deltas to attain other advantageous properties that were not addressed in the original version of the ★piler.

3.2 Variation 2

In its original design, the ★piler lacks guarantees regarding its transpilation capabilities. It simply attempts to find a viable transpilation from a starting parse tree; if successful, it implies the existence of a sequence of deltas performing the transpilation from that specific parse tree to the target grammar. What about other parse trees from the same grammar? It would be compelling to have guarantees about which parse trees can be transpiled in which target grammars. In this section, we address this challenge by introducing a mechanism to enhance delta predictability. The core concept is to imbue deltas with supplementary information. Previously, deltas only needed to uphold the semantics of the input parse tree. Here, we augment deltas with a form of type system. Each delta will specify which non-terminals it translates and what are the resulting introduced non-terminals. This augmentation aims to enhance predictability and constraints within the transpilation process.

Let us begin by defining the Non-Terminal Set (NTS).

Definition 3.2.1 (Non-Terminal Set). *Let $\Gamma = G_1, \ldots, G_M$ be a set of grammars. Let $N_1, \ldots, N_M$ the respective non-terminals. Let $N = N_1 \cup \cdots \cup N_M$. An NTS X of the grammar set Γ is any subset X of N, i.e., $X \in \mathcal{P}(N)$.*

Put differently, an NTS for specific grammars essentially constitutes a collection of non-terminals extracted from these grammars. The NTS functions as the tool that directs the transpilation process. With the NTS, we can specify which non-terminals a delta must translate and what the resulting non-terminals are. Let us delve into how NTS will be integrated within deltas to accomplish our objectives.

Definition 3.2.2 (NTS delta). *Let $\Gamma = \{G_1, \ldots, G_M\}$ be a set of grammars. Let A and B be NTS from Γ. An NTS delta, denoted $\delta^{A \to B} : \mathcal{T} \to \mathcal{T}$ (with signature $A \to B$), is a map such that for each $\tau \in \mathcal{T}$:*

1. $\forall \rho \in \mathrm{P}, \tau \in \mathcal{T} : [\![\tau]\!](\rho) = [\![\delta^{A \to B}(\tau)]\!](\rho)$
2. $\lambda(\delta^{A \to B}(\tau)) \cap (A - B) = \emptyset$
3. $\lambda(\delta^{A \to B}(\tau)) \cap B = B$

Furthermore, a delta $\delta^{A \to B}$ is applicable to the parse tree τ iff $\lambda(\tau) \cap A = A$.

According to the first criterion, an NTS delta, denoted as $\delta^{A \to B}$, must ensure the preservation of the semantics of the input parse tree (as with the ★piler). Regarding the other two properties, it is easier to consider them as subsequent application of two operations on the set $\lambda(\tau)$:

1. remove the elements of A from $\lambda(\tau)$.
2. add elements of B to $\lambda(\tau)$.

As one can easily check, applying these two operations does end in having $\lambda(\delta^{A \to B}(\tau)) \cap (A - B) = \emptyset$ and $\lambda(\delta^{A \to B}(\tau)) \cap B = B$. Furthermore, it is specified that a delta is applicable iff the node-set of the parse tree effectively contains all the non-terminals which the delta is supposed to translate (i.e., $\lambda(\tau) \cap A = A$). For clarity, let us consider an example. Let τ be a parse tree such that $\lambda(\tau) = \{a, b, c, d, e\}$ a set of non-terminals. Let $\delta^{\{a,c,e\} \to \{a,g\}}, \delta^{\{a,d\} \to \{e\}}$ be NTS deltas. If we apply the first delta to τ we obtain τ' such that $\lambda(\tau') = \{a, b, d, g\}$. From τ', we can apply only the second delta. The second delta yields a parse tree τ'' such that $\lambda(\tau'') = \{b, g, e\}$. Note that, neither the first nor the second delta are applicable on τ''.

Now, we extend a little the definition of NTS delta, introducing NTS+ deltas:

Definition 3.2.3 (NTS+ delta). *Let $\Gamma = \{G_1, \ldots, G_M\}$ be a set of grammars. Let $\{A_i\}_{i=1}^N$ and $\{B_i\}_{i=1}^N$ be NTS from Γ. An NTS+ delta, denoted $\delta^{A_1,\ldots,A_N \to B_1,\ldots,B_N} : \mathcal{T} \to \mathcal{T}$, is a map such that for each $\tau \in \mathcal{T}$:*

1. $\forall \rho \in \mathrm{P}, \tau \in \mathcal{T}_A : [\![\tau]\!](\rho) = [\![\delta^{A_1,\ldots,A_N \to B_1,\ldots,B_N}(\tau)]\!](\rho)$
2. $\forall i = \{1, \ldots, N\} \ : \ \lambda(\delta^{A_1,\ldots,A_i \to B_1,\ldots,B_i}(\tau)) \cap (A_i - B_i) = \emptyset.$
3. $\forall i = \{1, \ldots, N\} \ : \ \lambda(\delta^{A_1,\ldots,A_i \to B_1,\ldots,B_i}(\tau)) \cap B_i = B_i$

Furthermore, an NTS+ delta $\delta^{A_1,\ldots,A_N \to B_1,\ldots,B_N}$ is applicable to the parse tree τ iff

1. $\lambda(\tau) \cap A_1 = A_1$
2. $\forall i \in \{1, \ldots, N-1\} \; : \; \lambda(\delta^{A_1,\ldots,A_i \to B_1,\ldots,B_i}(\tau)) \cap A_{i+1} = A_{i+1}$

As you can see, the NTS+ delta is simply a generalization of the NTS delta. When $N = 1$ the NTS+ delta collapses to the previous definition of NTS delta. For sake of clarity, let us discuss the operations performed by an NTS+ delta such as $\delta^{A_1,\ldots,A_N \to B_1,\ldots,B_N}$ in terms of non-terminal sets:

- It removes elements from A_1.
- It adds elements from B_1.
- ...
- It removes elements from B_N.
- It adds elements from B_N.

This sequence of operations should meet the conditions 2) and 3) in the NTS+ definition. Furthermore, this delta is only applicable under a few conditions. Firstly, the interested parse tree must contain the non-terminals from A_1. Furthermore, whenever an A_i (with $1 < i \leq N$) is removed, we are removing exactly those elements (not one more, nor one less). The introduction of this generalization is necessary only because the notation NTS delta does not allow to determine the fact that composition of deltas remains a delta. However, this property holds for NTS+ deltas. This is the subject of the next remark.

Remark 3.2.1 (NTS+ delta composition). *Let $\Gamma = \{G_1, \ldots, G_M\}$ be a set of grammars. Let $\delta^{A_1,\ldots,A_N \to B_1,\ldots,B_N}$, $\delta^{C_1,\ldots,C_M \to D_1,\ldots,D_M}$ be NTS+ deltas. Then $\delta^{A_1,\ldots,A_N \to B_1,\ldots,B_N} \circ \delta^{C_1,\ldots,C_M \to D_1,\ldots,C_M}$ is an NTS+ delta with signature*

$$A_1, \ldots, A_N, C_1, \ldots, C_M \to B_1, \ldots, B_N, D_1, \ldots, D_M$$

.

Proof. Since both $\delta^{A_1,\ldots,A_N \to B_1,\ldots,B_N}$ and $\delta^{C_1,\ldots,C_M \to D_1,\ldots,C_M}$ do not change the semantics neither $\delta^{A_1,\ldots,A_N \to B_1,\ldots,B_N} \circ \delta^{C_1,\ldots,C_M \to D_1,\ldots,C_M}$ does (see Remark 2.2.1). Now consider what the subsequent application of these two deltas does in term of set operations, starting with $\delta^{A_1,\ldots,A_N \to B_1,\ldots,B_N}$:

- It removes elements from A_1.
- It adds elements from B_1.
- ...
- It removes elements from A_N.
- It adds elements from B_N.

Instead, $\delta^{C_1,\ldots,C_M \to D_1,\ldots,D_M}$:

- It removes elements from C_1.
- It adds elements from D_1.
- ...
- It removes elements from C_M.
- It adds elements from D_M.

The composition of $\delta^{A_1,\ldots,A_N \to B_1,\ldots,B_N}$ and $\delta^{C_1,\ldots,C_M \to D_1,\ldots,D_M}$ yields the concatenation of the previous operations which is exactly the definition of a delta with signature $A_1,\ldots,A_N,C_1,\ldots,C_M \to B_1,\ldots,B_N,D_1,\ldots,D_M$. Furthermore, the composed delta is only applicable when the concatenation of both deltas is applicable. Thus the composition of NTS+ deltas remains an NTS+ delta. □

Recall that both NTS and NTS+ deltas provide us with information about the non-terminals to be removed and added during the transpilation process. Furthermore, in the context of the ★piler, our primary focus was on transpilations that corresponded to solutions within the simplified search graph. This notion was represented as $\lambda(\vec{\delta}(\tau)) \subset N_G$, where τ denotes the initial parse tree and N_G (often referred to as the solution set) represents the set of non-terminals belonging to the target grammar G. By utilizing the distinctive characteristics of the newly introduced NTS+ deltas, we can strategically anticipate the potential transpilations that might lead to a solution. In order to facilitate this, let us define the following property:

Definition 3.2.4. *Let X and Y be NTSs. Let $\delta = \delta^{A_1,\ldots,A_N \to B_1,\ldots,B_N}$ be an NTS+ delta. We will say that δ moves X to Y, $X \xrightarrow{\delta} Y$ iff Y is the result of the following operations onto X: remove A_1, add B_1, ..., remove A_N, add B_N. Provided that $X \cap A_1 = A_1$, $((X - A_1) \cup B_1) \cap A_2 = A_2$, and so on.*

This specific property enables us to determine whether an NTS, denoted as Y, can be obtained by applying an NTS+ delta, represented as δ, to another NTS, referred to as X. It is important to note that NTS+ deltas are never directly applied to NTSs; rather, their application is always performed on parse trees. Nevertheless, one of the outcomes of applying NTS+ deltas to parse trees is the modification of the node-set based on their signature. Therefore, we can try to anticipate how the node-set of parse trees change based on the signatures of deltas. With this understanding in place, we can proceed to define the concept of an *NTS search graph*:

Definition 3.2.5 (NTS search graph)**.** *Let Δ be the set of NTS+ deltas. Let $\Gamma = \{G_1,\ldots,G_M\}$ be a set of grammars. Let $N = N_{G_1} \cup \cdots \cup N_{G_M}$ be the union of non-terminals. Let $V = \mathcal{P}(N)$. Let $E = \{(U,V) \in \mathcal{P}(N)^2 \mid \exists \delta \in \Delta \,.\, U \xrightarrow{\delta} V\}$. Then $S_{\Delta,\Gamma} = (V,E)$ is an NTS search graph.*

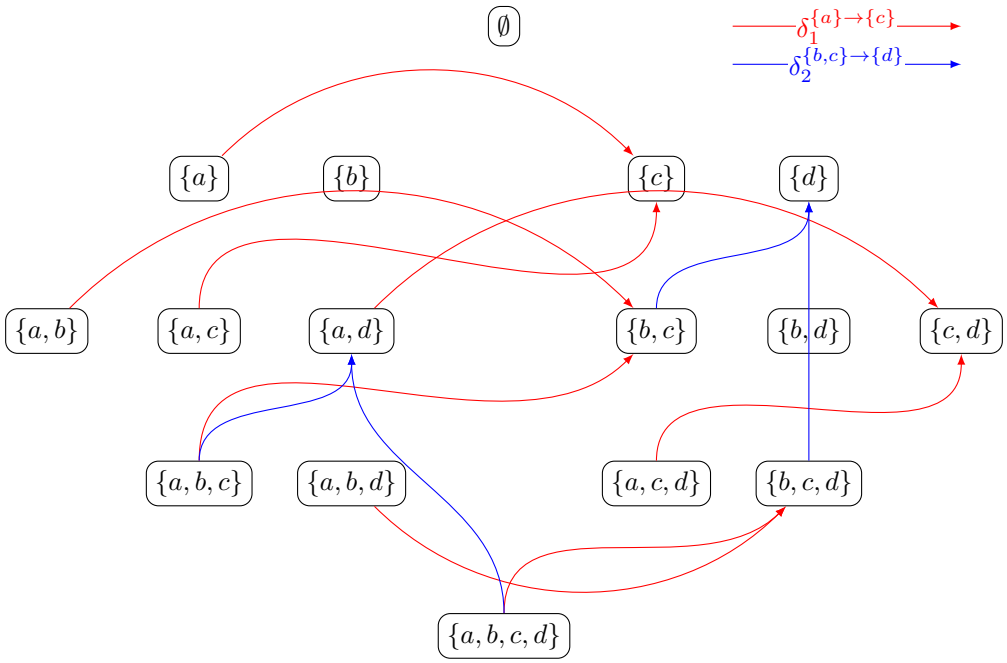


Figure 3.1. *An example of an NTS search graph with non-terminals $N = \{a, b, c, d\}$ and two deltas with signatures $\{a\} \to \{c\}$ and $\{b, c\} \to \{d\}$. Edges from the first delta are colored in red. Edges from the second delta are colored in blue.*

An NTS search graph is essentially a graph that emerges from the consideration of all potential applications of NTS+ deltas (Δ) across every conceivable node-set, denoted as $\mathcal{P}(N)$. Given that the number of nodes is exponential in relation to the number of non-terminals ($|\mathcal{P}(N)| = 2^{|N|}$), such a graph naturally becomes immensely large. However, it is important to note that we do not need to store the entire graph. Instead, we can dynamically compute the specific segments of the graph that require exploration. Fig. 3.1 provides a visual example. Here, the possible nodes are $\mathcal{P}(\{a, b, c, d\})$ (16 nodes). Edges are generated starting from two deltas with signatures $\{a\} \to \{c\}$ and $\{b, c\} \to \{d\}$ respectively. As one can see, even with very few non-terminals and only two deltas the graph becomes quite large and densely connected. Fortunately, there is no need to completely generate or explore such a graph. We can simply explore only those paths that are relevant for the task at hand. For example, consider a starting grammar with non-terminals $\{a, b\}$ and a target grammar with non-terminals $\{c, d\}$. We know that every parse tree, τ, that we will have to transpile will have non-terminals from set $\{a, b\}$ ($\lambda(\tau) \subseteq \{a, b\}$). Therefore, if we receive a parse tree such that $\lambda(\tau) = \{a, b\}$, we already know that the only admissible solution is given by application of $\vec{\delta}_{12} = \delta_1 \circ \delta_2$. Notice that, it is not guaranteed that such a path is a solution. However, as we will see, we are guaranteed that if a solution exists then we will find it. For completeness, we should also consider the case in which the source parse tree, τ, is such that $\lambda(\tau) = \{a\}$ or $\lambda(\tau) = \{b\}$. In the first case, $\vec{\delta}_3 = \delta_1$, reach a solution. While, in the second case, it is not able to reach a solution. To translate

such a parse tree, we need a delta that can be applied to a parse tree with a node set containing exclusively b. Without introducing new deltas, we can deduce that any parse tree τ such that $\lambda(\tau) = \{b\}$ cannot be transpiled to an equivalent parse tree having non-terminals from the target grammar. In order to formalize these concepts we introduce the *NTS search problem*.

Definition 3.2.6 (NTS search problem). *The search problem is defined by the triple* $(S_{\Delta,\Gamma} = (V, E), S, T)$*, where* $S_{\Delta,\Gamma}$ *is an NTS search graph,* S *and* T *are the source and target NTS respectively.*

The first main difference wrt. the original approach is the lack of the starting parse tree (τ) and target grammar. Here, τ is no longer needed. Now, knowing the signatures of deltas we can precompute paths that lead from one NTS to another. In other words, we can precompute for all parse trees τ such that $\lambda(\tau) = S$ the possible transpilations $\vec{\delta}$ that yield the target NTS (such that $\vec{\delta}(\tau) \subseteq T$). This consideration also leads us to the definition of NTS search solution.

Definition 3.2.7 (NTS search solution). *Given the NTS search problem* $(S_{\Delta,\Gamma} = (V, E), S, T)$*, a composition of NTS deltas* $\delta_0, \circ, \ldots, \circ, \delta_n = \vec{\delta}$*, such that* $\delta_i \in \Delta$*, is a solution iff* $S \xrightarrow{\vec{\delta}} X$*, such that* $X \subseteq T$*.*

In other words, a composition of deltas is a solution iff, by considering their signatures, their composition yields an NTS set that is a subset of T starting from the source NTS S. Notice that this time there is no need to worry about the transpilation exiting the NTS search graph $S_{\Delta,\Gamma}$, since the $S_{\Delta,\Gamma}$ is generated by considering the $\mathcal{P}(N)$ and all possible applications of NTS deltas in Δ.

Now, let us consider a practical scenario. Consider a source grammar G_S and a target grammar G_T. Suppose we want to transpile parse trees from G_S to parse trees of G_T, thus we set up an NTS search problem. Now, the NTS search problem requires a source NTS and a target NTS. We can easily fix the target NTS with N_{G_T} (the non-terminals of the target grammar). However, how should we choose the source NTS? Consider that if we fix the source NTS to be N_{G_S}, we will find solutions only for the parse tree τ such that $\lambda(\tau) = N_{G_S}$. Thus, in order to precompute all possible solutions for all possible parse trees we need to consider all subsets of N_{G_S}. Unfortunately, this means exploring a graph $2^{|N_{G_S}|}$ times! This is unfeasible even for relatively small grammars. On the other hand, we can still search this graph on the fly, and a search on such a graph is much faster compared to the ★piler searches, as traversing an edge involves only set operations. Furthermore, we can also cache the solutions found on the NTS search problem and we would initiate a new graph exploration only upon the change of the problem (i.e. either the search graph or the source NTS or the target NTS changes).

We should also consider another aspect. Suppose we found a set of solutions for a particular search problem (G, S, T). Thus, we have found transpilations $\vec{\delta}$ such that: $\lambda(\tau) = S \implies \lambda(\vec{\delta}(\tau)) \subseteq T$. While we know that τ is semantically equivalent to $\vec{\delta}(\tau)$, it may not be true that $\vec{\delta}(\tau) \lhd G_T$. This is a fairly strange event, we obtained a

semantically equivalent transpilation with all the required non-terminals that is not compliant with the target grammar. Unfortunately, we do not have a practical example of a delta that could cause such an event. Nonetheless, recall that this same situation was faced by the ★piler in its original framework. Furthermore, we addressed this problem in Sect. 3.1 by introducing further requirements on the deltas. Thus, one could combine the approaches (Variant described in Sect. 3.1 and Variant described in Sect. 3.2) to obtain the best of both worlds.

Finally, to conclude this section, we need to show that if there is a transpilation for parse tree τ to the target grammar G_T. Then this transpilation is contained in the set of solutions for the NTS search problem $(S_{\Delta,\Gamma}, \lambda(T), N_{G_T})$. In order to prove this result, we will reintroduce the concept of search graph/problem and solution. We will need to show that a solution for the search problem is a solution for the NTS search problem.

Definition 3.2.8 (search graph). *Let $\Delta = \{\delta_0, \ldots, \delta_N\}$ be a set of NTS+ deltas. Let $V \subseteq \mathcal{T}_\Gamma$ be a set of parse trees from grammars $\Gamma = \{G_0, \ldots, G_M\}$. Let $E = \{(\tau_1, \tau_2) \in V \times V \mid \exists\, \delta \in \Delta \,.\, \delta(\tau_1) = \tau_2\}$. We call $S_{\Delta,\Gamma} = (V, E)$ the search graph.*

Definition 3.2.9 (search problem). *The search problem is defined by the triple $(S_{\Delta,\Gamma} = (V, E), \tau, G)$, where $S_{\Delta,\Gamma}$ is a search graph, τ is a starting parse tree from V and G is a target grammar from Γ.*

Definition 3.2.10 (search solution). *Given the search problem $(S_{\Delta,\Gamma} = (V, E), \tau, G)$, a composition of deltas $\vec{\delta}_I$ is a solution when:*

1. $\vec{\delta}_I(\tau) \lhd G$.
2. $\forall i \geq 0 : \vec{\delta}_{I[:i]}(\tau) \in V$.
3. $\forall i \geq 0 : (\vec{\delta}_{I[:i]}(\tau),\ \vec{\delta}_{I[:i+1]}(\tau)) \in E$.

Now, these three definition are identical to the ones proposed for the original ★piler framework. The only difference is the usage of NTS+ deltas instead of the traditional deltas. Note that, for each search problem $(S_{\Delta,\Gamma} = (V, E), \tau, G)$, we can build the respective NTS search problem $(S'_{\Delta,\Gamma} = (V, E), \lambda(\tau), N_G)$. Using these definitions, we can show that a solution for a search problem is also a solution for the NTS search problem. This is the subject of the following theorem:

Theorem 3.2.1. *Let $\vec{\delta}$ be a solution for the search problem $(S_{\Delta,\Gamma}, \tau, G)$, then $\vec{\delta}$ is a solution for the respective NTS search problem $(S'_{\Delta,\Gamma}, \lambda(\tau), N_G)$.*

Proof. The proof trivially follows by the first property of search solution (i.e., $\vec{\delta}(\tau) \lhd G$). Since $\vec{\delta}(\tau) \lhd G$, we have that $\lambda(\vec{\delta}(\tau)) \subseteq N_G$. □

Thus if a solution exists for a specific search problem it also exists for the respective NTS search problem. This particular property allows us to search only solutions for the NTS search problem and check those solutions against the original problem. The advantage of using the NTS domain is twofold. for once 1) we can cache solutions, as

the set of used non-terminals from a parse tree is not likely to change much across compilations. 2) Simulating a transpilation using the signatures is much faster than using the deltas directly on parse trees. By using the signature, we only need to either add or remove elements from a set until we reach a solution. Instead, by using the delta directly, we may incur in heavy operations that traverse large parse trees. Furthermore, we will show that, it is possible to use A* to search solution for NTS search problem. Firstly, we need to define a metric space for $\mathcal{P}(N)$, i.e., we need to define a metric space for the nodes of the NTS search graph. Luckily, we have already encountered the distance function that provide such a metric space, d_{sdd}.

Lemma 3.2.1. *Let N be the union of non-terminals from grammars $G_1, \ldots, G_N$. Then $(\mathcal{P}(N), d_{sdd})$ is a metric space.*

Proof. The proof trivially follows by the fact that d_{sdd} is a distance function on sets (shown in Definition 2.2.10), such as those contained in $\mathcal{P}(N)$. □

Now, we only lack a proper non-overestimating heuristic so that A* can be used.

Definition 3.2.11. *Let T be the solution NTS. Let $h_T : \mathcal{P}(N) \to \mathbb{R}$ be a non-negative map such that:*

$$\bar{h}_T(X) = \begin{cases} 0 & \text{if } X \subseteq T \\ \min_{Y \subseteq T} d_{sdd}(X, Y) & \text{otw.} \end{cases} \tag{3.2}$$

Theorem 3.2.2. *$\bar{h}^T$ is an admissible heuristic (i.e., non-overestimating).*

Proof. Let (G, S, T) be the generic NTS problem at hand. Let $T^* \subseteq T$ be a generic target node. Let $\vec{\delta}_I = \delta_1 \circ \cdots \circ \delta_k$ (for $k \geq 1$) be an NTS delta such that $S \xrightarrow{\vec{\delta}_I} T^*$. We need to show that

$$\bar{h}_T(S) \leq \sum_{i=1}^{k-1} d_{sdd}(\lambda(\vec{\delta}_I[:i](\tau)), \lambda(\vec{\delta}_I[:i+1](\tau))$$

In other words, we need to show that the distance traveled by a generic chain of NTS+ deltas within the NTS search graph is always greater or equal to the distance estimated by our heuristic. We proceed by case.

- ($S \subseteq T$). If $S \subseteq T \implies \bar{h}_T(S) = 0$ which does not overestimate.
- (otherwise). $\bar{h}_T(S) \leq d_{sdd}(S, T^*)$. Now, consider our situation in the graph. We start at node S and we want travel to node T^*. In order to do so, we need to travel a sequence of deltas. If there is a single delta that brings us to the destination T^*, then we traveled $d_{sdd}(S, T^*)$ which is clearly greater than or equal to $\min_{Y \subseteq T} d_{sdd}(S, Y)$. Thus, we do not overestimate over paths of a single delta. Now suppose that we need to travel a sequence of deltas greater than 1. Then, we start at S, we travel to a middle node B and then we travel again to the solution T^*. By triangular inequality (since the graph is embedded in a metric sapce), we are traveling more distance than going directly from S to T^*. Since $\bar{h}_T(S) \leq d_{sdd}(S, T^*)$, we are not overestimating.

□

Finally, let us recap the overall steps on which the second variant of the ★piler is built. Recall that we added to deltas preconditions and postconditions. The precondition tells which non-terminals are removed (or translated) and the postconditions tells what non-terminals are added as a result of the delta application. These preconditions and postconditions can be used to induce a graph from the set of nodes represented by non-terminals sets ($V = \mathcal{P}(N)$). Now, recall that if a transpilation brings a source parse tree τ in a parse tree τ' for a target grammar G (i.e., $\tau' \lhd G$) then $\lambda(\tau') \subseteq N_G$. Thus such a path must exist in the NTS search graph. Therefore, by enumerating solutions in the NTS search graph, we are guaranteed to find a solution if one exist.

3.3 Variation 3

Recall that the first variation aimed to remove the necessity of repeated searches in the ★piler framework. This necessity arose from the fact that even when a parse tree node-set ($\lambda(\tau)$) agrees with the set of non-terminals for a target grammar ($\lambda(\tau) \subseteq G$), we cannot be sure that the achieved parse tree is compliant with the target grammar ($\tau \lhd G$ need not be true). In Sect. 3.1, we solved this issue by introducing the augmented deltas, that is, we enforced the fact that if $\lambda(\delta(\tau)) \subseteq N_G \implies \delta(\tau) \lhd G$. The rest of Sect. 3.1 is then dedicated to show that the variation works as much as the ★piler does.

Now, recall also that the second variation aimed to enhance the search of the induced graph by avoiding the direct application of deltas. The idea is to introduce additional information on the delta (precondintions and postconditions) so that the search graph can be generated and explored ahead of time. The Sect. 3.2 is entirely dedicated to this approach.

Both these variations arise by modifying the definition of deltas. If we combine the previous definitions (augmented delta and NTS+ delta), we can introduce the NTS+ augmented delta which recites:

Definition 3.3.1 (NTS+ augmented delta). *Let $G_1, \ldots, G_M$ be a set of grammars. Let N_i be the set of non-terminals of grammar G_i. An augmented delta, denoted $\delta_+^{A\to B}$, is a map $\delta_+^{A\to B} : \mathcal{T} \to \mathcal{T}$ such that:*

1. $\forall \rho \in \mathrm{P}, \tau \in \mathcal{T} : [\![\tau]\!](\rho) = [\![\delta_+^{A\to B}(\tau)]\!](\rho)$
2. $\lambda(\delta_+^{A\to B}(\tau)) \subseteq N_i \implies \delta^+(\tau) \lhd G_i$
3. $\lambda(\delta_+^{A\to B}(\tau)) \cap (A - B) = \emptyset$
4. $\lambda(\delta_+^{A\to B}(\tau)) \cap B = B$

Furthermore, we say that a delta $\delta_+^{A\to B}$ is applicable to the parse tree τ iff $\lambda(\tau) \cap A = A$.

The definition of NTS+ augmented delta generates a variation of the ★piler that combines both benefits of the previous variations. However, developing a transpilation

function that fits the definition of NTS+ augmented deltas becomes increasingly difficult. Of course, in practice, one can very rarely even prove that a transpilation function does not change the semantics of the input which is a core requirement not only for the ★piler but for all transpilers in general.

4
Related Work

In this work, we developed the theoretical and practical framework called the ★piler. The ★piler is an exotic transpilation infrastructure that aims to improve reusability of language components and reducing glue code. However, during the years, the research community worked tirelessly proposing a variety of approaches addressing the same issues addressed by the ★piler. In this section, we provide an overview of the most promising approaches.

In particular, in Sect. 4.1, we will delve into the topic of Language Workbenches, and in Sect. 4.2, we will explore the concept of Language Product Lines (LPLs).

4.1 Language Workbenches

As you might have gathered, crafting programming languages is a challenging endeavor. It demands comprehension of intricate components and advanced programming skills. Moreover, delving into optimizations, creating development environments, and devising debugging tools significantly amplifies the complexity of language development.

The development of programming languages or domain-specific languages holds immense value as these languages serve as our interfaces with machines. Furthermore, the choice of interface can vary based on the task at hand. For instance, when describing data for storage and manipulation, dedicated syntaxes like JSON [1] or YAML [2] might be ideal. In cases involving interaction with databases, SQL or its implementations [3] might be preferred. Similarly, for crafting web pages, HTML [4] could be the option to go for. This diversity underscores how languages are tailored to simplify interactions with computers, adapting to distinct needs.

To address this complexity, the research community introduced the concept of Language Workbenches. These workbenches offer comprehensive tool sets for constructing, modifying, and overseeing programming languages and their associated tools. They provide an integrated environment for language engineering, facilitating the development of languages, domain-specific languages, integrated development environments (IDEs), and compilers.

During the years, the research community proposed several language workbenches,

[1] `www.json.org`
[2] `yaml.org`
[3] `www.postgresql.org`
[4] `html.spec.whatwg.org`

encompassing very different ideas. For a more complete discussion of reusability in language workbenches, we refer to [7].

Melange Melange [27, 26, 24] serves as a language workbench that seamlessly integrates tools from the Eclipse Modeling Framework (EMF) ecosystem [80]. Within Melange, abstract syntaxes are defined through Xtext, while corresponding semantics are specified via Xtend-based Kermeta 3 aspects. Constructing a language within Melange involves creating an Ecore model that declaratively outlines a set of classes for the abstract syntax, complemented by Kermeta semantic aspects. These aspects are intricately linked to Ecore-generated classes via specific annotations. Within Erdweg *et al.*'s classification [28], Melange accommodates language extension, unification, self-extension, and extension composition [56]. The workbench boasts sophisticated composition mechanisms like the extension of existing languages or amalgamation of multiple languages [27].

MontiCore MontiCore [33, 45, 72] stands as a distinctive language workbench that employs a singular domain-specific language for the definition of both abstract and concrete syntax. This workbench generates class models automatically and their semantics are realized in Java through abstract syntax tree (AST) visitors. These visitors gain access to inherited and synthesized grammar attributes via an injected getter/setter API integrated into the target Abstract Data Types (ADTs). In accordance with the classification by Erdweg *et al.* [28], MontiCore covers language extension, unification, self-extension, and extension composition [56]. ADTs linked to AST nodes can be expanded to recycle existing semantics, and the framework supports multiple grammar inheritance.

Meta Programming System (MPS) MPS [89, 66] emerges as a development environment tailored for non-textual domain-specific languages. Unlike traditional approaches, MPS adopts a *projectional* [90] paradigm, circumventing the need for parsers. The framework defines abstract syntax through meta-models, stored in non-human-editable XML files. Instead of traditional code, programs are modeled by composing elemental building blocks called *concepts*, each representing a type of AST node. Various components like *behavior* and *editors* can be linked to these concepts to construct both the visual representation and the semantics of the AST using a subset of Java known as `BaseLanguage`. In alignment with Erdweg *et al.*'s classification [28], MPS encompasses language extension, unification, self-extension, and extension composition [56].

Neverlang Neverlang [11, 17, 84, 16] introduces a distinctive language workbench focused on modular development of programming language compilers, interpreters, and their related ecosystems. Central to the Neverlang approach is the concept of *language features*. These features are realized through compilation units termed *slices*, which in turn implement them by composing various *modules*. This composition process is *syntax-driven*, where the language grammar dictates insertion points for semantic

actions. Modules can also carry meta-data, contributing to the development of the language ecosystem, including IDE support. The Automatic Integrated Development Environment (AiDE) complements Neverlang, catering to LPL development [48, 47, 30]. AiDE serves three roles: language developer, deployer, and user, accommodating all phases of language family development. In accordance with the taxonomy by Erdweg *et al.* [28], Neverlang supports language extension, unification, self-extension, and extension composition [56].

Spoofax Spoofax [91, 88, 40] stands as a language workbench offering diverse DSLs for language development. A significant feature is its use of Syntax Definition Formalism (SDF3) to specify grammars, which may even be ambiguous, and Stratego for semantics—a sequence of AST transformations called *rules* and *strategies*. The program abstract syntax is maintained as a ATerm data structure. As per the Erdweg *et al.* classification [28], Spoofax facilitates language extension, unification, self-extension, and extension composition [56].

Rascal Rascal [41, 4, 42] functions as a meta-programming language dedicated to constructing language processing tools. Algebraic data types are used to define the abstract syntax, and Rascal library facilitates parsing textual input, followed by *implosion* for tree transformations. For evaluating ASTs, user-defined functions and pattern matching of algebraic data types on function arguments are used, employing a technique called *pattern-based dispatch* [4].

CBS CBS [22, 59, 58] stands as a language workbench partly developed with Rascal, designed for component-based programming language and DSL development. The central idea of CBS is the utilization of *funcons*, modular components that can be repurposed across various language specifications [23, 9]. The PlanCompS initiative [10] aspires to establish a comprehensive funcon library, serving as a foundation for language development within CBS.

Truffle Truffle [93, 99, 50] is a library aimed at building language interpreters and implementations. This approach involves abstract syntax tree node rewriting to carry out optimizations and semantic operations. By coupling Truffle with GraalVM [98, 97], successful programming language implementations have emerged across diverse projects [61, 43].

Compared to these existing tools, the ★piler represents a drastically different approach, and it comes with its own set of advantages and drawbacks. Many of the established tools in this domain have benefited from years of development efforts, making them more mature and feature-rich. These tools often provide extensive support for the development of integrated development environments (IDEs) and other language-related tools.

In contrast, the approach adopted by the ★piler is distinctive in that it effectively minimizes the need for glue code when reusing language components. This reduction in glue code is achieved through a compilation/transpilation process that involves a search step, which, notably, only requires the parse tree and the transpilation units. While the ★piler may not offer the same level of maturity or feature richness as some of these other tools, its approach has the potential to streamline the process of language component reuse.

4.2 Language Product Lines

LPLs [56] represent an innovative engineering approach to language development and customization. These product lines are designed to facilitate the creation of families of domain-specific languages or programming languages with shared characteristics (such as role base languages [49]). LPLs enable developers to efficiently generate and maintain multiple language variants within a coherent framework. By embracing concepts from software product lines, LPLs empower researchers and practitioners to build tailored languages that cater to specific needs and application domains.

LPLs are aimed at modeling families of programming languages where each member, referred to as a variant, can be tailored to satisfy specific domain or customer requirements. One notable example of successful LPL engineering is Neverlang.js [48, 14, 15]. Neverlang.js represents a family of JavaScript-like programming languages that have been designed to gradually introduce language features to computer science students.

Many language workbenches, such as Neverlang, also adopt the philosophy of LPL engineering as an approach to enhance reusability and address language variability [86, 41]. For a comprehensive overview of LPLs and language workbenches, you can refer to [29]. Additionally, there are tools like AiDE, which is a variability management tool for LPLs [85, 86]. AiDE operates as a bottom-up LPL engineering environment [47], helping manage the complexity of language families through the utilization of Feature Models [39, 25, 82] and language configurators with automatic constraint checking. While there is no best configarators, researchers have designed tools to model the variability of configurators [8, 95].

Since LPLs represent an engineering methodology for the development of programming languages, researchers have also focused on evaluating the quality of such LPLs. This evaluation includes the proposal of a set of software metrics [13]. Moreover, there is a focus on mutation operators for mutation testing of LPLs, as discussed in [12].

4.3 Transpilers

Transpilers are software that aims to translate one program written in one language into a different one. For example, SequalsK [74] is a bidirectional transpiler between Swift and Kotlin. SequalsK aims to bridge the development of Android and IOS applications. Ling *et al.* [53] have developed a C to Rust source-to-source transpiler with the purpose of migrating old source code. Also, a partial Python to Rust transpiler

is proposed by Lunnikivi *et al.* [55]. They have shown a 12x performance improvement in the transpiled code. These transpilers are only concerned with a single source language and a single target language (one-to-one). Instead the ★piler framework offers a homogeneous approach to develop many-to-many transpilers. ROSE [68, 69] is a compiler infrastructure with an intermediate representation that supports a variety of languages, such as C, C++, and Java. The ROSE infrastructure uses tree transformations to apply optimizations on the intermediate representation. EpsilonFlock [71, 70] is a tool developed using the Epsilon platform [44] to perform a rule based meta-model transpilation. Instead, of using an intermediate representation to which all languages need to be transpiled to, the ★piler framework allows a level of flexibility that permits the development of both one-to-one transpilers and one-to-many transpilers. Moreover, delta developed for any transpilers can always be reused for different scenarios.

4.4 Multi-Language Systems

Folliot *et al.* [31] describe a multi-language system—called VVM. VVM is a virtual machine running VMlets which can execute the bytecode of their language. Self [96] is a minimal programming language that is optimized during runtime. Self has been used to implement languages such as Java and Smalltalk without relying on custom VMs. VMs usually run a single intermediate language. However, they can be effective at unifying the ecosystem of the languages that compiles toward a VM (successful examples are the Java and the Scala programming languages). Instead, the ★piler framework can be effective also at unifying the ecosystem of languages that are developed to run on different VMs. Moreover, the ★piler framework can decouple a language from its VM. It can allow to run the same language on different VMs.

Postface

We are finally arrived at the conclusion of this thesis. I sincerely hope that the material presented throughout these pages has been easily understandable for the reader, as few things are more frustrating than encountering unclear explanations. To address any potential shortcomings on my part in delivering the subject matter effectively, I wholeheartedly invite readers to reach out with any questions or concerns via mail to francesco.bertolotti@unimi.it. Furthermore, I trust that readers have found the content engaging and even worthy of further investigations. However, if this is not the case, as the saying goes: "it is what it is".

Bibliography

[1] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. `code2seq`: Generating Sequences from Structured Representations of Code. In Alexander Rush, editor, *Proceedings of the 7th International Conference on Learning Representations (ICLR'19)*, New Orleans, LA, USA, May 2019.

[2] John W Backus, Friedrich L Bauer, Julien Green, Charles Katz, John McCarthy, Alan J Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, Joseph Henry Wegstein, et al. Report on the algorithmic language algol 60. *Communications of the ACM*, 3(5):299–314, 1960.

[3] Aditi Barthwal and Michael Norrish. Verified, Executable Parsing. In Giuseppe Castagna, editor, *Proceedings of the 18th European Symposium on Programming (ESOP'09)*, Lecture Notes in Computer Science 5502, pages 160–174, York, United Kingdom, March 2009. Springer.

[4] Bas Basten, Jeroen van den Bos, Mark Hills, Paul Klint, Arnold Lankamp, Bert Lisser, Atze van der Ploeg, Tijs van der Storm, and Jurgen Vinju. Modular Language Implementation in Rascal—Experience Report. *Science of Computer Programming*, 114:7–19, December 2015.

[5] Francesco Bertolotti and Walter Cazzola. CombTransformers: Statement-Wise Transformers for Statement-Wise Representations. *IEEE Transactions on Software Engineering*, 49(10):4677–4690, October 2023.

[6] Francesco Bertolotti and Walter Cazzola. Fold2Vec: Towards a Statement Based Representation of Code for Code Comprehension. *Transaction on Software Engineering and Methodology*, 32(1):6:1–6:31, February 2023.

[7] Francesco Bertolotti, Walter Cazzola, and Luca Favalli. On the Granularity of Linguistic Reuse. *Journal of Systems and Software*, 202, August 2023.

[8] Francesco Bertolotti, Walter Cazzola, and Luca Favalli. SPLLSP: Software Product Lines Extraction Driven by Language Server Protocol. *Journal of Systems and Software*, 205, November 2023.

[9] L. Thomas van Binsbergen, Peter D. Mosses, and Neil Sculthorpe. Executable Component-Based Semantics. *Journal of Logical and Algebraic Methods in Programming*, 103(2):184–212, February 2019.

[10] L. Thomas van Binsbergen, Neil Sculthorpe, and Peter D. Mosses. Tool Support for Component-Based Semantics. In *Companion Proceedings of the 15th International Conference on Modularity (Companion Modularity'16)*, pages 8–11, Málaga, Spain, March 2016. ACM.

[11] Walter Cazzola. Domain-Specific Languages in Few Steps: The Neverlang Approach. In Thomas Gschwind, Flavio De Paoli, Volker Gruhn, and Matthias Book, editors, *Proceedings of the 11th International Conference on Software Composition (SC'12)*, Lecture Notes in Computer Science 7306, pages 162–177, Prague, Czech Republic, 31st of May-1st of June 2012. Springer.

[12] Walter Cazzola, Francesco Cesarini, and Luca Tansini. PerformERL: A Performance Testing Framework for Erlang. *Distributed Computing*, 35:439–454, May 2022.

[13] Walter Cazzola and Luca Favalli. Towards a Recipe for Language Decomposition: Quality Assessment of Language Product Lines. *Empirical Software Engineering*, 27(4), April 2022.

[14] Walter Cazzola and Diego Mathias Olivares. Gradually Learning Programming Supported by a Growable Programming Language. In *Proceedings of the IEEE 39th Annual Computer Software and Applications Conference (COMPSAC'15)*, page 857, Taichung, Taiwan, 1st-5th of July 2015. IEEE. Extended Abstract.

[15] Walter Cazzola and Diego Mathias Olivares. Gradually Learning Programming Supported by a Growable Programming Language. *IEEE Transactions on Emerging Topics in Computing*, 4(3):404–415, September 2016. Special Issue on Emerging Trends in Education.

[16] Walter Cazzola and Albert Shaqiri. Context-Aware Software Variability through Adaptable Interpreters. *IEEE Software*, 34(6):83–88, November 2017. Special Issue on Context Variability Modeling.

[17] Walter Cazzola and Edoardo Vacchi. Neverlang 2: Componentised Language Development for the JVM. In Walter Binder, Eric Bodden, and Welf Löwe, editors, *Proceedings of the 12th International Conference on Software Composition (SC'13)*, Lecture Notes in Computer Science 8088, pages 17–32, Budapest, Hungary, 19th of June 2013. Springer.

[18] Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José Luis Marroquín. Searching in metric spaces. *ACM computing surveys (CSUR)*, 33(3):273–321, 2001.

[19] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. A Survey of Compiler Testing. *ACM Computing Surveys*, 53(1):4:1–4:36, February 2021.

[20] Noam Chomsky. Three models for the description of language. *IRE Transactions on information theory*, 2(3):113–124, 1956.

[21] Noam Chomsky. *Syntactic structures*. Mouton de Gruyter, 2002.

[22] Martin Churchill, Peter D. Mosses, Neil Schulthorpe, and Paolo Torrini. Reusable Components of Semantic Specifications. *Transaction on Aspect-Oriented Software Development*, 12(1):132–179, March 2015.

[23] Martin Churchill, Peter D. Mosses, and Paolo Torrini. Reusable Components of Semantic Specifications. In Eric Ernst, editor, *Proceedings of the 13th International Conference on Modularity (Modularity'14)*, pages 145–156, Lugano, Switzerland, April 2014. ACM.

[24] Benoît Combemale. *Towards Language-Oriented Modeling*. Habilitation À Diriger Des Recherches, Université de Rennes 1, Rennes, France, December 2015.

[25] Krysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In Sven Apel and Stefania Gnesi, editors, *Proceedings of the 6th International Workshop on Variability Modeling of Software-Intensive Systems (VaMoS'12)*, pages 173–182, Leipzig, Germany, January 2012. ACM.

[26] Thomas Degueule. Interoperability and Composition of DSLs with Melange. ACM Student Research Competition Grand Finals, June 2016.

[27] Thomas Degueule, Benoît Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. Melange: a Meta-Language for Modular and Reusable Development of DSLs. In Davide Di Ruscio and Markus Völter, editors, *Proceedings of the 8th International Conference on Software Language Engineering (SLE'15)*, pages 25–36, Pittsburgh, PA, USA, October 2015. ACM.

[28] Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann Rendel. Language Composition Untangled. In Anthony M. Sloane and Suzana Andova, editors, *Proceedings of the 12th Workshop on Language Description, Tools, and Applications (LDTA'12)*, Tallinn, Estonia, March 2012. ACM.

[29] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Alex Kelly, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. Evaluating and Comparing Language Workbenches: Existing Results and Benchmarks for the Future. *Computer Languages, Systems and Structures*, 44:24–47, December 2015.

[30] Luca Favalli, Thomas Kühn, and Walter Cazzola. Neverlang and FeatureIDE Just Married: Integrated Language Product Line Development Environment. In Philippe Collet and Sarah Nadi, editors, *Proceedings of the 24th International Software Product Line Conference (SPLC'20)*, pages 285–295, Montréal, Canada, 19th-23rd of October 2020. ACM.

[31] Berti Folliot, Ian Piumarta, and Fabio Riccardi. A Dynamically Configurable, Multi-Language Execution Platform. In Paulo Guedes and Jean Bacon, editors, *Proceedings of the 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications (EW'98)*, pages 175–181, Sintra, Portugal, September 1998. ACM.

[32] Sebastian Götz, Thomas Kühn, Christian Piechnick, Georg Püschel, and Uwe Aßmann. A models@ run. time approach for multi-objective self-optimizing software. In *International Conference on Adaptive and Intelligent Systems*, pages 100–109. Springer, 2014.

[33] Hans Grönninger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore: A Framework for the Development of Textual Domain Specific Languages. In Wilhelm Schäfer, Matthew Dwyer, and Volker Gruhn, editors, *Companion Proceedings of the 30th International Conference on Software Enginering (Companion ICSE'08)*, pages 925–926, Leipzig, Germany, May 2008. IEEE.

[34] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, July 1968.

[35] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1):60–65, 2001.

[36] K.J. Horadam and M.A. Nyblom. Distances between Sets Based on Set Commonality. *Discrete Applied Mathematics*, 167:310–314, April 2014.

[37] Gerhard Jäger and James Rogers. Formal language theory: refining the chomsky hierarchy. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 367(1598):1956–1970, 2012.

[38] Rahul Kala, Anupam Shukla, and Ritu Tiwari. Fusion of probabilistic a* algorithm and fuzzy inference system for robotic path planning. *Artificial Intelligence Review*, 33:307–327, 2010.

[39] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, November 1990.

[40] Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. Pure and Declarative Syntax Definition: Paradise Lost and Regained. In *Proceedings of ACM Conference on New Ideas in Programming and Reflections on Software (Onward! 2010)*, Reno-Tahoe, Nevada, USA, October 2010. ACM.

[41] Paul Klint, Tijs van der Storm, and Jurgen Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In Andrew Walenstein and Sibylle Schupp, editors, *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM'09)*, pages 168–177, Edmonton, Canada, September 2009. IEEE.

[42] Paul Klint, Tijs van der Storm, and Jurgen Vinju. Rascal, 10 Years Later. In *Proceedings of the 19th International Working Conference on Source Code Analysis and*

Manipulation (SCAM'19), pages 139–139, Cleveland, OH, USA, September/October 2019. IEEE.

[43] Sebastian Kloibhofer, Thomas Pointhuber, Maximilian Heisinger, Hanspeter Mössenböck, Lukas Stadler, and David Leopoldseder. SymJEx: Symbolic Execution on te GraalVM. In Stefan Marr, editor, *Proceedings of the 17th International Conference on Managed Programming Languages and Runtimes (MPLR'20)*, pages 63–72. ACM, November 2020.

[44] Dimitris Kolovos. *An Extensible Platform for Specification of Integrated Languages for Model Management*. Phd thesis, University of York, York, United Kingdom, 2008.

[45] Holger Krahn, Bernhard Rumpe, and Steven Völkel. MontiCore: A Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer*, 12(5):353–372, September 2010.

[46] Nikhil Krishnaswamy. Comparison of efficiency in pathfinding algorithms in game development. *Technical Reports*, 10, 2009.

[47] Thomas Kühn and Walter Cazzola. Apples and Oranges: Comparing Top-Down and Bottom-Up Language Product Lines. In Rick Rabiser and Bing Xie, editors, *Proceedings of the 20th International Software Product Line Conference (SPLC'16)*, pages 50–59, Beijing, China, 19th-23rd of September 2016. ACM.

[48] Thomas Kühn, Walter Cazzola, and Diego Mathias Olivares. Choosy and Picky: Configuration of Language Product Lines. In Goetz Botterweck and Jules White, editors, *Proceedings of the 19th International Software Product Line Conference (SPLC'15)*, pages 71–80, Nashville, TN, USA, 20th-24th of July 2015. ACM.

[49] Thomas Kühn, Max Leuthäuser, Sebastian Götz, Christoph Seidl, and Uwe Aßmann. A metamodel family for role-based modeling and programming languages. In *Software Language Engineering: 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings 7*, pages 141–160. Springer, 2014.

[50] Florian Latifi, David Leopoldseder, Cristian Wimmer, and Hanspeter Mössenböck. CompGen: Generation of Fast JIT Compilers in a Multi-Language VM. In Arjun Guha, editor, *Proceedings of the 17th International Symposium on Dynamic Languages (DLS'21)*, pages 35–47, Chicago, IL, USA, October 2021. ACM.

[51] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In Michael D. Smith, editor, *Proceedings of the 2nd International Symposium on Code Generation and Optimization (CGO'04)*, pages 75–86, San José, CA, USA, March 2004. IEEE.

[52] Willem JM Levelt. *An introduction to the theory of formal languages and automata*. John Benjamins Publishing, 2008.

[53] Michael Ling, Yijun Yu, Haitao Wu, Yuan Wang, James R. Cordy, and Ahmed E. Hassan. In Rust We Trust: A Transpiler from Unsafe C to Safer Rust. In *Companion Proceedings of the 44th International Conference on Software Engineering (ICSE'22-Companion)*, pages 354–355, Pittsburgh, PA, USA, May 2022. IEEE.

[54] Peter Linz and Susan H. Rodger. *An introduction to formal languages and automata*. Jones & Bartlett Learning, 2022.

[55] Henri Lunnikivi, Kai Jylkkä, and Timo Hämäläinen. Transpiling Python to Rust for Optimized Performance. In Alex Orailoglu, Matthias Jung, and Marc Reichenbach, editors, *Proceedings of the 20th International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS'20)*, Lecture Notes in Computer Science 12471, pages 127–138, Samos, Greece, July 2020. Springer.

[56] David Méndez-Acuña, José A. Galindo, Thomas Degueule, Benoît Combemale, and Benoît Baudry. Leveraging Software Product Lines Engineering in the Development of External DSLs: A Systematic Literature Review. *Computer Languages, Systems & Structures*, 46:206–235, November 2016.

[57] Johannes Mey, Thomas Kühn, René Schöne, and Uwe Assmann. Reusing static analysis across different domain-specific languages using reference attribute grammars. *The Art, Science, and Engineering of Programming*, 4(3):15–1, 2020.

[58] Peter D. Mosses. A Component-Based Formal Language Workbench. In Rosemary Monahan, Virgile Prevosto, and José Proença, editors, *Proceedings of the 5th Workshop on Formal Integrated Development Environment (F-IDE'19)*, pages 29–34, Porto, Portugal, October 2019.

[59] Peter D. Mosses. Software Meta-Language Engineering and CBS. *Journal of Computer Languages*, 50:39–48, February 2019.

[60] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications: an appetizer*. Springer Science & Business Media, 2007.

[61] Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. GraalSqueak: Toward a Smalltalk-Based Tooling Platform for Polyglot Programming. In Irene Finocchi, editor, *Proceedings of the 16th International Conference on Managed Programming Languages and Runtimes (MPLR'19)*, pages 14–26, Athens, Greece, October 2019. ACM.

[62] Noraimi Azlin Mohd Nordin, Norhidayah Kadir, Zati Aqmar Zaharudin, and Nor Amalina Nordin. An application of the a* algorithm on the ambulance routing. In *2011 IEEE Colloquium on Humanities, Science and Engineering*, pages 855–859. IEEE, 2011.

[63] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An Extensible Compiler Framework for Java. In *Proceedings of the 12th International Conference on Compiler Construction (CC'03)*, Lecture Notes in Computer Science 2622, pages 138–152, Warsaw, Poland, April 2003. Springer.

[64] Anish Paranjpe and Gang Tan. Bohemia–a validator for parser frameworks. In *2021 IEEE Security and Privacy Workshops (SPW)*, pages 162–170. IEEE, 2021.

[65] Terence J. Parr and Russell W. Quong. ANTLR: A Predicated-LL(k) Parser Generator. *Software—Practice and Experience*, 25(7):789–810, July 1995.

[66] Vaclav Pech, Alex Shatalin, and Markus Völter. JetBrains MPS as a Tool for Extending Java. In Walter Binder, editor, *Proceedings of the 10th International Conference on Principles and Practices of Programming on the Java Platform (PPPJ'13)*, pages 165–168, Stuttgart, Germany, September 2013. ACM.

[67] Gordon D Plotkin. *A structural approach to operational semantics*. Aarhus university, 1981.

[68] Dan Quinlan. ROSE: Compiler Support for Object-Oriented Frameworks. *Parallel Processing Letters*, 10(2):215–226, June 2000.

[69] Dan Quinlan and Chunhua Liao. The Rose Source-to-Source Compiler Infrastructure. In Sam Midkiff, Rudi Eigenmann, and Hansang Bae, editors, *Proceedings of the Cetus Users and Compiler Infrastructure Workshop*, pages 1–3, Galveston, TX, USA, October 2011.

[70] Louis M. Rose, Dimitrios Kolovos, Richard F. Paige, Fiona A. C. Polack, and Simon Poulding. Epsilon Flock: A Model Migration Language. *Software and Systems Modeling*, 13(2):735–755, May 2014.

[71] Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. Model Migration with Epsilon Flock. In Laurence Tratt and Martin Gogolla, editors, *Proceedings of the 3rd International Conference on Theory and Practice of Model Transformations (ICMT'10)*, Lecture Notes in Computer Science 6142, pages 184–198, Malága, Spain, June 2010. Springer.

[72] Bernhard Rumpe, Katrin Hölldobler, and Oliver Kautz. MontiCore: Language Workbench and Library. Handbook, Aachen, Germany, March 2021.

[73] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. Feature diagrams: A survey and a formal semantics. In *14th IEEE International Requirements Engineering Conference (RE'06)*, pages 139–148. IEEE, 2006.

[74] Dominik Schultes. SequalsK—A Bidirectional Swift-Kotlin-Transpiler. In Rui Abreu and Mattia Fazzini, editors, *Proceedings of the 8th International Conference on Mobile Software Engineering and Systems (MobileSoft'21)*, pages 73–83, Madrid, Spain, May 2021. IEEE.

[75] Dana Scott. *Outline of a mathematical theory of computation*. Oxford University Computing Laboratory, Programming Research Group Oxford, 1970.

[76] Dana S Scott and Christopher Strachey. *Toward a mathematical semantics for computer languages*, volume 1. Oxford University Computing Laboratory, Programming Research Group Oxford, 1971.

[77] Roger S Scowen. Generic base standards. In *Proceedings 1993 Software Engineering Standards Symposium*, pages 25–34. IEEE, 1993.

[78] Jeffrey Shallit. *A second Course in Formal Languages And Automata Theory*. Cambridge University, 2009.

[79] Michael Sipser. Introduction to the theory of computation. *ACM Sigact News*, 27(1):27–29, 1996.

[80] Dave Steinberg, Dave Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, December 2008.

[81] Chengnian Sun, Vu Le, and Zhendong Su. Finding Compiler Bugs via Live Code Mutation. In Yannis Smaragdakis, editor, *Proceedings of the 31st International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16)*, pages 849–863, Amsterdam, Netherlands, November 2016. ACM.

[82] Thomas Thüm, Leopoldo Teixeira, Klaus Schmid, Eric Walkingshaw, Mukelabai Mukelabai, Mahsa Varshosaz, Goetz Botterweck, Ina Schaefer, and Timo Kehrer. Towards efficient analysis of variation in time and space. In *Proceedings of the 23rd International Systems and Software Product Line Conference-Volume B*, pages 57–64, 2019.

[83] Masaru Tomita. *Generalized LR parsing*. Springer Science & Business Media, 1991.

[84] Edoardo Vacchi and Walter Cazzola. Neverlang: A Framework for Feature-Oriented Language Development. *Computer Languages, Systems & Structures*, 43(3):1–40, October 2015.

[85] Edoardo Vacchi, Walter Cazzola, Benoît Combemale, and Mathieu Acher. Automating Variability Model Inference for Component-Based Language Implementations. In Patrick Heymans and Julia Rubin, editors, *Proceedings of the 18th International Software Product Line Conference (SPLC'14)*, pages 167–176, Florence, Italy, 15th-19th of September 2014. ACM.

[86] Edoardo Vacchi, Walter Cazzola, Suresh Pillay, and Benoît Combemale. Variability Support in Domain-Specific Language Development. In Martin Erwig, Richard F. Paige, and Eric Van Wyk, editors, *Proceedings of 6th International Conference on Software Language Engineering (SLE'13)*, Lecture Notes on Computer Science 8225, pages 76–95, Indianapolis, USA, 27th-28th of October 2013. Springer.

[87] Tijs van der Storm and Felienne Hermans. Gradual grammars: Syntax in levels and locales. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering*, pages 134–147, 2022.

[88] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart CL Kats, Eelco Visser, and GH Wachsmuth. Dsl engineering-designing, implementing and using domain-specific languages. 2013.

[89] Markus Völter and Vaclav Pech. Language Modularity with the MPS Language Workbench. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*, pages 1449–1450, Zürich, Switzerland, June 2012. IEEE.

[90] Markus Völter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. Towards User-Friendly Projectional Editors. In Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju, editors, *Proceedings of the 7th International Conference on Software Language Engineering (SLE'14)*, Lecture Notes in Computer Science Volume 8706, pages 41–61, Västerås, Sweden, September 2014. Springer.

[91] Guido H. Wachsmuth, Gabriël D. P. Konat, and Eelco Visser. Language Design with the Spoofax Language Workbench. *IEEE Software*, 31(5):35–43, September/October 2014.

[92] Patrick Wang. Pseutopy: Towards a non-english natural programming language. In *Proceedings of the 17th ACM Conference on International Computing Education Research*, pages 429–430, 2021.

[93] Christian Wimmer and Thomas Würthinger. Truffle: A Self-Optimizing Runtime System. In Gary T. Leavens, editor, *Proceedings of the 3rd Annual Conference on Systems, Programming and Applications: Software for Humanity (SPLASH'12)*, pages 1–2, Tucson, AZ, USA, October 2012. ACM.

[94] Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT press, 1993.

[95] Jan Willem Wittler, Thomas Kühn, and Ralf Reussner. Towards an integrated approach for managing the variability and evolution of both software and hardware components. In *Proceedings of the 26th ACM International Systems and Software Product Line Conference (SPLC'22)-Volume B*, pages 94–98, 2022.

[96] Mario Wolczko, Agesen. Ole, and David Ungar. Towards a Universal Implementation Substrate for Object-Oriented Languages. In *Proceedings of the OOPSLA Workshop on Simplicity, Performance and Portability in Virtual Machine Design (WSPPVMD'99)*, Denver, CO, USA, November 1999.

[97] Thomas Würthinger. Graal and Truffle: Modularity and Separation of Concerns as Cornerstones for Building a Multipurpose Runtime. In *Companion Proceedings of the 13th International Conference on Modularity (Companion Modularity'14)*, pages 3–4, Lugano, Switzerland, April 2014. ACM.

[98] Thomas Würthinger, Christian Wimmer, Andreas Woß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to Rule Them All. In Robert Hirschfeld, editor, *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!'13)*, pages 187–204, Indianapolis, IN, USA, October 2013. ACM.

[99] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-Optimizing AST Interpreters. In Alessandro Warth, editor, *Proceedings of the 8th Symposium on Dynamic languages (DSL'12)*, pages 73–82, Tucson, AZ, USA, October 2012. ACM.

[100] Huilai Zou, Lili Zong, Hua Liu, Chaonan Wang, Zening Qu, and Youtian Qu. Optimized application and practice of a* algorithm in game map path-finding. In *2010 10th IEEE International Conference on Computer and Information Technology*, pages 2138–2142. IEEE, 2010.